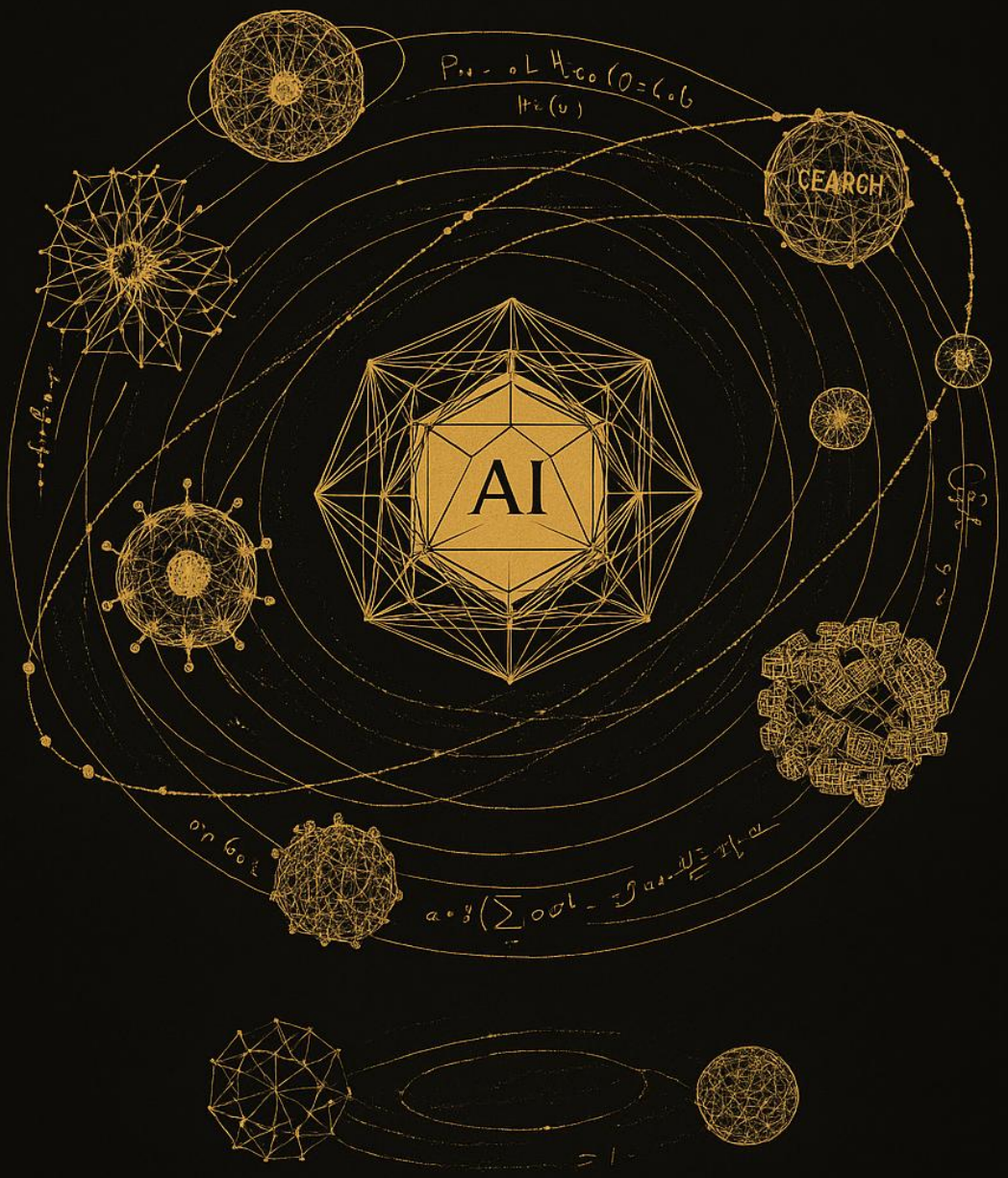
АЛГОРИТМИ І СТРУКТУРИ ДАНИХ

BYCHKOVIVS OLEXIVS



Алгоритми і структури даних

проєкт першого підручника

О.С.Бичков

ЧАСТИНА І: ФУНДАМЕНТАЛЬНІ КОНЦЕПЦІЇ ТА ЛІНІЙНІ СТРУКТУРИ 29

Розділ 1: Вступ та аналіз алгоритмів 29
1.1. Поняття алгоритму, структури даних та абстрактного типу даних (ADT). 29
1. Поняття алгоритму 29
1.1. Історія виникнення та суть поняття 29
1.2. Визначення та властивості алгоритму 30
1.3. Способи опису алгоритмів 31
1.4. Класичний приклад: алгоритм Евкліда 32
1.5. Математичний аналіз алгоритмів і асимптотична нотація 33
1.6. Класифікація алгоритмів за парадигмами 33
1.7. Життєвий цикл алгоритму — від ідеї до оптимізації 34
2. Структури даних 36
2.1. Визначення та потреба 36
2.2. Класифікація структур даних 36
2.3. Детальний огляд основних структур даних 37
2.4. Як вибір структури даних впливає на швидкодію 39
2.5. Розширений каталог структур даних та їх асимптотики 40
2.5. Спеціалізовані структури даних для зовнішньої пам’яті 41
2.6. Архітектурні аспекти: кеші та локальність посилань 43
2.7. Перевірені шаблони вибору структури даних 43
3. Абстрактний тип даних (АТД): поглиблений погляд 44
3.1. Ідея абстракції: «Що» проти «Як» 44
3.2. Формальне визначення та компоненти АТД 44
3.3. АТД проти Структура Даних: у чому різниця? 45
3.4. Сила абстракції: навіщо потрібні АТД? 46
3.5. Дизайн за контрактом (Design by Contract) та формальна верифікація АТД 47
3.6. Приклад повної специфікації: АТД «Множина» (Set) 47
3.7. Параметричний поліморфізм і generic-контейнери 48
1.2. Аналіз складності алгоритмів 48
Переваги та недоліки асимптотичного аналізу 61
Контрольні запитання 61
Тести 63
Завдання для самостійної роботи 65

Базові лінійні структури диних 68
Порядок та Послідовність як основа організації даних 68
Дві філософії зберігання: суцільність проти зв’язності 69
Від зберігання до поведінки: Стеки та Черги як Абстракції 71
Що чекає на вас у цьому розділі? 72
2.1. Масив: статичний та динамічний 72
Контрольні запитання 77
Методичні вказівки до лабораторної роботи 77
Тести 79
Тести підвищеної складності 80
Завдання для самостійної роботи 82
2.2. Рядок. 83
Контрольні запитання 88

Методичні вказівки до лабораторної роботи ... 89
7. Тести ... 90
Тести підвищеної складності ... 91
Завдання для самостійної роботи ... 93
Зв'язаний список ... 94
Переваги та недоліки ... 102
Контрольні запитання ... 103
Методичні вказівки до лабораторної роботи ... 103
Тести ... 105
Тести підвищеної складності ... 106
Завдання для самостійної роботи ... 108
Стек ... 109
Реалізація Стеку на основі масиву ... 110
Реалізація Стеку на основі зв'язного списку ... 112
Переваги та недоліки ... 114
Контрольні запитання ... 114
Методичні вказівки до лабораторної роботи ... 115
Тести ... 116
Тести підвищеної складності ... 117
Завдання для самостійної роботи ... 119
Черга, кільцева черга. ... 120
Переваги та недоліки ... 125
Контрольні запитання ... 125
Методичні вказівки до лабораторної роботи ... 125
Тести ... 127
Тести підвищеної складності ... 128
Завдання для самостійної роботи ... 130
Дек ... 131
Переваги та недоліки ... 136
Контрольні запитання ... 136
Методичні вказівки до лабораторної роботи ... 137
Тести ... 138
Тести підвищеної складності ... 139
Завдання для самостійної роботи ... 141
Множина, мультимножина, мішок як абстрактні типи. ... 142
Теоретичні відомості ... 142
АТД "Множина" (Set): Контракт та Операції ... 143
Реалізації Множини: Швидкість проти Порядку ... 144
Мультимножина (Multiset) та Мішок (Bag) ... 145
Переваги та недоліки ... 146
Контрольні запитання ... 147
Методичні вказівки до лабораторної роботи ... 147
Тести ... 148
Тести підвищеної складності ... 149
Завдання для самостійної роботи ... 152

ЧАСТИНА II: АЛГОРИТМИ ПОШУКУ ... 154

Лінійні алгоритми пошуку ... 154
Лінійний пошук ... 155
Теоретичні відомості ... 155
Псевдокод та його аналіз ... 156

Аналіз складності: найкращий, середній та найгірший випадки ... 157
Область застосування ... 157
Приклади застосування ... 158
Переваги та недоліки ... 158
Контрольні запитання ... 159
Методичні вказівки до лабораторної роботи ... 159
Тести ... 160
Тести підвищеної складності ... 161
Завдання для самостійної роботи ... 163
Бінарний пошук ... 164
Теоретичні відомості ... 164
Ітеративна реалізація ... 165
Рекурсивна реалізація ... 166
Аналіз складності: сила логарифму ... 167
Переваги та недоліки ... 168
Контрольні запитання ... 169
Методичні вказівки до лабораторної роботи ... 169
Тести ... 170
Тести підвищеної складності ... 171
Завдання для самостійної роботи ... 174
Пошук з бар'єром ... 175
Переваги та недоліки ... 178
Контрольні запитання ... 179
Методичні вказівки до лабораторної роботи ... 179
Тести ... 180
Тести підвищеної складності ... 181
Завдання для самостійної роботи ... 184

Інші методи пошуку ... 184
Пошук стрибками ... 184
Переваги та недоліки ... 187
Контрольні запитання ... 188
Методичні вказівки до лабораторної роботи ... 188
Тести ... 189
Тести підвищеної складності ... 190
Завдання для самостійної роботи ... 192
Інтерполяційний пошук ... 193
Переваги та недоліки ... 197
Контрольні запитання ... 197
Методичні вказівки до лабораторної роботи ... 197
Тести ... 199
Тести підвищеної складності ... 200
Завдання для самостійної роботи ... 202
Експоненційний пошук ... 203
Переваги та недоліки ... 207
Контрольні запитання ... 207
Методичні вказівки до лабораторної роботи ... 208
Тести ... 209
Тести підвищеної складності ... 210
Завдання для самостійної роботи ... 212

Гібридні алгоритми пошуку ... 213
Інтерполяційно-двійковий пошук ... 213

Переваги та недоліки ... 217
Контрольні запитання ... 217
Методичні вказівки до лабораторної роботи ... 217
Тести ... 219
Тести підвищеної складності ... 220
Завдання для самостійної роботи ... 222
Гібридний двійково-лінійний пошук ... 223
Переваги та недоліки ... 227
Контрольні запитання ... 227
Методичні вказівки до лабораторної роботи ... 227
Тести ... 228
Тести підвищеної складності ... 229
Завдання для самостійної роботи ... 232
Пошук з режимом галопу ... 233
Переваги та недоліки ... 236
Контрольні запитання ... 236
Методичні вказівки до лабораторної роботи ... 237
Тести (прості) ... 238
Тести підвищеної складності ... 239
Завдання для самостійної роботи ... 241

Ефективний пошук: хешування та дерева ... 242

Хеш-структури ... 243
Хеш-таблиці: ... 244
Переваги та недоліки ... 249
Контрольні запитання ... 249
Методичні вказівки до лабораторної роботи ... 250
Тести ... 251
Тести підвищеної складності ... 252
Завдання для самостійної роботи ... 254

Імовірнісні фільтри ... 255
Фільтр Блума ... 255
Переваги та недоліки ... 259
Контрольні запитання ... 259
6. Методичні вказівки до лабораторної роботи ... 260
Тести ... 261
Тести підвищеної складності ... 262
Завдання для самостійної роботи ... 265
Фільтр Блума, що рахує ... 266
Переваги та недоліки ... 270
Контрольні запитання ... 270
Методичні вказівки до лабораторної роботи ... 270
Тести ... 272
Тести підвищеної складності ... 273
Завдання для самостійної роботи ... 275
Фільтр Зозулі ... 276
Переваги та недоліки ... 281
Контрольні запитання ... 281
Методичні вказівки до лабораторної роботи ... 282
Тести ... 283
Тести підвищеної складності ... 284

Завдання для самостійної роботи 287
Фільтр-частка 288
Переваги та недоліки 292
Контрольні запитання 292
Методичні вказівки до лабораторної роботи 293
Тести 294
Тести підвищеної складності 295
Завдання для самостійної роботи 297

Дерева пошуку 299
Дерева пошуку 299
Бінарні дерева пошуку 301
Процедура видалення вузла з бінарного дерева пошуку 305
Обходи дерева 308
Пошук в глибину (DFS) та його варіації 308
Пошук в ширину (BFS) або обхід по рівнях 321
Аналіз складності операцій у бінарному дереві пошуку 332
Переваги та недоліки 335
Контрольні запитання 335
Методичні вказівки до лабораторної роботи 335
Тести BST 336
Тести підвищеної складності BST 337
Завдання для самостійної роботи 340
Збалансовані бінарні дерева пошуку 341
AVL-дерево 343
Механізм балансування: обертання 344
Аналіз складності операцій в AVL-дереві 350
Переваги та недоліки 352
Контрольні запитання 352
Методичні вказівки до лабораторної роботи 352
Тести 354
Тести підвищеної складності 355
Завдання для самостійної роботи 357
Червоно-чорне дерево 358
Переваги та недоліки 373
Контрольні запитання 373
Методичні вказівки до лабораторної роботи 373
Тести 375
Тести підвищеної складності 376
Завдання для самостійної роботи 378
Splay-дерево (Розширюване дерево) 379
Переваги та недоліки 396
Методичні вказівки до лабораторної роботи 397
Тести 398
Тести підвищеної складності 399
Завдання для самостійної роботи 402
Декартове дерево 403
Операція Вставки 409
Операція Видалення 411
Пошук та обходи 413
Аналіз складності операцій 413
Переваги та недоліки 416
Контрольні запитання 417

Методичні вказівки до лабораторної роботи 417
Тести 418
Тести підвищеної складності 419
Завдання для самостійної роботи 422
Дерева з розгалуженням > 2 423
Дерева з розгалуженням > 2: 2-3-4 та (a,b)-дерева 424
Аналіз складності операцій у 2-3-4 та (a,b)-деревах 430
Переваги та недоліки 433
Контрольні запитання 433
Методичні вказівки до лабораторної роботи 433
Тести 435
Тести підвищеної складності 436
Завдання для самостійної роботи 438
B-дерева: B-дерево, B+-дерево, B*-дерево 439
Операції у класичному B-дереві 442
Обхід B дерева 447
Аналіз складності операцій у класичному B-дереві 450
Операції у B+-дереві 452
Обхід B+-дерева 458
Аналіз складності операцій у B+-дереві 461
Операції у B*-дереві 463
Обхід B*-дерева 467
Аналіз складності операцій у B*-дереві 469
Переваги та недоліки 472
Контрольні запитання 475
Методичні вказівки до лабораторної роботи 475
Тести 477
Тести підвищеної складності 478
Завдання для самостійної роботи 480
Дерево порядкових статистик 481
Структура вузла OST 483
Переваги та недоліки 501
Контрольні запитання 501
Методичні вказівки до лабораторної роботи 502
Тести 503
Тести підвищеної складності 504
Завдання для самостійної роботи 507
Лабораторні роботи: Order-Statistic Trees 508
Функціональність: 509

Купи та пріоритетні черги 517
Бінарна купа 518
Абстрактний тип даних “Черга з пріоритетами” 533
Переваги та недоліки 536
Контрольні питання 537
Методичні рекомендації до виконання лабораторної роботи 537
Тести 539
Тестові завдання підвищеної складності 540
Завдання для самостійної роботи 542

Біноміальні купи 543
Переваги та недоліки 575
Недоліки: 575

Контрольні запитання ... 576
Методичні вказівки до лабораторної роботи: Біноміальні купи ... 577
Тести ... 583
Тести підвищеної складності ... 584

Купа Фібоначчі ... 587
Операції Купи Фібоначчі ... 589
Обходи Дерева ... 609
Переваги та Недоліки ... 617
Контрольні Запитання ... 619
Методичні вказівки до лабораторних робіт ... 619
Тести ... 622
Тести підвищеної складності ... 624
Завдання для самостійної роботи ... 627

Подвійна купа ... 628
Переваги та Недоліки ... 650
Контрольні Запитання ... 651
Методичні вказівки до лабораторної роботи ... 652
Тести ... 655
Тести підвищеної складності ... 656
Завдання для самостійної роботи ... 659

Купа лівостороння ... 660
Переваги та Недоліки ... 685
Контрольні Запитання ... 686
Методичні вказівки до лабораторної роботи: ... 686
Тести ... 690
Тести підвищеної складності ... 691
Завдання для самостійної роботи ... 694

Скошена купа ... 695
Переваги та Недоліки ... 714
Контрольні запитання ... 715
Методичні Вказівки до лабораторної роботи ... 716
Тести ... 719
Тести підвищеної складності ... 720
Завдання для самостійної роботи ... 724

ЧАСТИНА III: АЛГОРИТМІЧНІ ПАРАДИГМИ ... 725

Розділ 6: Класичні парадигми проєктування ... 725
1. Розділяй і володарюй ... 725
Алгоритм множення Карацуби ... 725
Переваги та недоліки ... 728
Контрольні запитання ... 729
Методичні вказівки до лабораторної роботи ... 729
Тести ... 731
Завдання для самостійної роботи ... 732
Алгоритм множення Тоома-Кука ... 733
Переваги та недоліки ... 737
Контрольні запитання ... 738
Методичні вказівки до лабораторної роботи ... 738

Тести 739
Завдання для самостійної роботи 740
Алгоритм множення Шенхаге-Штрассена 741
Переваги та недоліки 745
Контрольні запитання 745
Методичні вказівки до лабораторної роботи 745
Тести 747
Завдання для самостійної роботи 748
Алгоритм множення матриць Штрассена 749
Переваги та недоліки 753
Контрольні запитання 754
Методичні вказівки до лабораторної роботи 754
Тести 756
Завдання для самостійної роботи 757
Жадібні алгоритми 758
Переваги та недоліки 762
Контрольні запитання 762
Методичні вказівки до лабораторної роботи 762
Тести 764
Завдання для самостійної роботи 765
Динамічне програмування: 767
Переваги та недоліки 771
Контрольні запитання 772
Методичні вказівки до лабораторної роботи 772
Тести 773
Завдання для самостійної роботи 774
Динамічне програмування в задачі «Найдовша спільна підпослідовність» 776
Переваги та недоліки 780
Контрольні запитання 781
Методичні вказівки до лабораторної роботи 781
Тести 782
Завдання для самостійної роботи 783
Динамічне програмування в задачі «Редакційна відстань» 785
Переваги та недоліки 789
Контрольні запитання 789
Методичні вказівки до лабораторної роботи 789
Тести 790
Завдання для самостійної роботи 791
Динамічне програмування на підмножинах 793
Переваги та недоліки 797
Контрольні запитання 798
Методичні вказівки до лабораторної роботи 798
Тести 799
Завдання для самостійної роботи 800
Динамічне програмування по розрядах 802
Переваги та недоліки 806
Контрольні запитання 806
Методичні вказівки до лабораторної роботи 807
Тести 808
Завдання для самостійної роботи 809
Пошук з поверненням 810
Переваги та недоліки 815
Контрольні запитання 815

Методичні вказівки до лабораторної роботи ... 816
Тести ... 817
Завдання для самостійної роботи ... 818
Метод гілок та меж ... 819
Переваги та недоліки ... 823
Контрольні запитання ... 824
Методичні вказівки до лабораторної роботи ... 824
Тести ... 826
Завдання для самостійної роботи ... 827

ЧАСТИНА IV: АЛГОРИТМИ НА ГРАФАХ ... 829

Графи: Фундаментальна структура даних ... 829
Обходи графів ... 844
Застосування Обходів Графів ... 852
1. Компоненти зв'язності ... 853
2. Мости ... 857
3. Точки Шарніру ... 863
4. Топологічне Сортування ... 866
5. Пошук Циклів ... 875
Контрольні Запитання ... 881
Методичні вказівки до лабораторної роботи ... 882
Тести ... 886
Тести підвищеної складності ... 888
Завдання для самостійної роботи ... 891
Теми лабораторних робіт ... 892
Сильно зв'язні компоненти (SCC): алгоритми Косараджу, Тар'яна. ... 894
Контрольні запитання ... 911
Методичні вказівки до лабораторної роботи ... 911
Тести ... 916
Тести підвищеної складності ... 917
Завдання для самостійної роботи ... 920
Теми лабораторних робіт ... 922
Ейлерові та Гамільтонові шляхи/цикли. ... 924
Контрольні Запитання ... 937
Методичні вказівки до лабораторної роботи ... 938
Тести ... 942
Тести підвищеної складності ... 944
Завдання для самостійної роботи ... 947
Тем лабораторних робіт ... 948
Алгоритм Ієрхольцера для знаходження Ейлерових шляхів/циклів ... 948
Контрольні Запитання ... 957
Тести ... 958
Тести підвищеної складності ... 959
Завдання для самостійної роботи ... 962

Найкоротші шляхи, потоки та паросполучення ... 964
Найкоротші шляхи (одна вершина) ... 964

Алгоритм Дейкстри для знаходження найкоротших шляхів ... 964
Контрольні Запитання ... 975
Методичні вказівки до лабораторної роботи ... 976

Тести 980
Тести підвищеної складності 981
Завдання для самостійної роботи 984

Алгоритм Беллмана-Форда для знаходження найкоротшого шляху 985
Контрольні запитання 994
Методичні вказівки до лабораторної роботи 995
Тести 1000
Тести підвищеної складності 1001
Завдання для самостійної роботи 1004

Алгоритм SPFA (Shortest Path Faster Algorithm) 1005
Контрольні запитання 1014
Методичні вказівки до лабораторної роботи 1015
Тести 1019
Завдання для самостійної роботи 1023
Найкоротші шляхи (всі пари) 1025
Алгоритм Флойд-Воршелла 1025
Контрольні Запитання 1035
Методичні вказівки до лабораторної роботи 1035
Тести 1039
Тести підвищеної складності 1041
Завдання для самостійної роботи 1044
Алгоритм Джонсона 1045
Контрольні Запитання 1053
Методичні вказівки до лабораторної роботи 1054
Тести 1058
Тести підвищеної складності 1060
Завдання для самостійної роботи 1062
Евристичний пошук: A* 1064
Контрольні запитання 1077
Методичні вказівки до лабораторної роботи 1078
Тести 1082
Тести підвищеної складності 1083
Завдання для самостійної роботи 1086
Мінімальні кістякові дерева (MST): алгоритми Крускала, Прима, Борувки. 1087
Алгоритм Прима для пошуку мінімального кістякового дерева 1088
Контрольні запитання 1097
Методичні вказівки до лабораторної роботи 1098
Тести 1100
Тести підвищеної складності 1102
Завдання для самостійної роботи 1106
Теми лабораторних робіт 1106
Алгоритм Крускала та структури даних “система неперетинних множин” 1107
Контрольні запитання 1116
Методичні вказівки до лабораторної роботи 1116
Тести 1119
Тести підвищеної складності 1120
Завдання для самостійної роботи 1124
Теми лабораторних робіт 1125
Максимальний потік / Мінімальний розріз. 1126
Максимальний потік. Метод Форда-Фалкерсона 1127
Методичні вказівки до лабораторної роботи 1135

Тести ... 1137
Тести підвищеної складності ... 1139
Завдання для самостійної роботи ... 1142
Теми лабораторних робіт ... 1143
Максимальний потік. Алгоритм Едмондса-Карпа ... 1144
Контрольні запитання ... 1150
Методичні вказівки до лабораторної роботи ... 1151
Тести ... 1153
Тести підвищеної складності ... 1154
Завдання для самостійної роботи ... 1158
Теми лабораторних робіт ... 1159
Максимальний потік. Алгоритм Дініца ... 1160
Переваги та недоліки ... 1165
Контрольні запитання ... 1166
Методичні вказівки до лабораторної роботи ... 1166
Тести ... 1169
Тести підвищеної складності ... 1170
Завдання для самостійної роботи ... 1173
Теми лабораторних робіт ... 1174
Максимальний потік. Метод проштовхування–підняття мітки ... 1175
Контрольні запитання ... 1182
Методичні вказівки до лабораторної роботи ... 1182
Тести ... 1184
Тести підвищеної складності ... 1185
Завдання для самостійної роботи ... 1188
Теми лабораторних робіт ... 1189
Паросполучення ... 1191
Паросполучення в дводольних графах. Алгоритм Гопкрофта-Карпа ... 1192
Контрольні запитання ... 1198
Методичні вказівки до лабораторної роботи ... 1199
Тести підвищеної складності ... 1202
Завдання для самостійної роботи ... 1206
Теми лабораторних робіт ... 1207
Паросполучення в загальних графах. Алгоритм Блоссом ... 1207
Контрольні запитання ... 1214
Методичні вказівки до лабораторної роботи ... 1215
Тести підвищеної складності ... 1218
Відповіді на тести підвищеної складності ... 1220
Завдання для самостійної роботи ... 1222
Теми лабораторних робіт ... 1222
Система неперетинних множин ... 1223
Оптимізація “Об’єднання за рангом” ... 1224
Переваги та недоліки ... 1229
Контрольні запитання ... 1230
Методичні вказівки до лабораторної роботи ... 1230
Тести ... 1232
Тести підвищеної складності ... 1234
Завдання для самостійної роботи ... 1237
Оптимізація “Стиснення шляху” ... 1239
Переваги та недоліки ... 1243
Контрольні запитання ... 1243
Тести ... 1245
Тести підвищеної складності ... 1246

Завдання для самостійної роботи 1250
Теми лабораторних робіт 1251

Частина V: Спеціалізовані алгоритми та структури 1251

Розділ 9: Сортування та порядкові статистики 1251
9.1. Прості сортування 1251
9.1.1 Бульбашкове сортування 1251
Контрольні запитання 1256
Методичні вказівки до лабораторної роботи 1256
Тести 1258
Тести підвищеної складності 1259
Завдання для самостійної роботи 1263
Теми лабораторних робіт 1263
9.1.2 Сортування вибором 1264
Контрольні запитання 1269
Методичні вказівки до лабораторної роботи 1269
Тести 1271
Тести підвищеної складності 1272
Завдання для самостійної роботи 1275
Теми лабораторних робіт 1276
9.1.3 Сортування вставкою 1277
Контрольні запитання 1281
Методичні вказівки до лабораторної роботи 1281
Тести 1283
Тести підвищеної складності 1284
Завдання для самостійної роботи 1287
Теми лабораторних робіт 1288
9.1.4 Сортування Шелла 1289
Контрольні запитання 1293
Методичні вказівки до лабораторної роботи 1294
Тести 1296
Тести підвищеної складності 1297
Завдання для самостійної роботи 1300
Теми лабораторних робіт 1300
9.1.5 Шейкерне сортування 1301
Контрольні запитання 1306
Методичні вказівки до лабораторної роботи 1306
Тести 1307
Тести підвищеної складності 1309
Завдання для самостійної роботи 1311
Теми лабораторних робіт 1312
9.1.6 Сортування гнома 1313
Контрольні запитання 1317
Методичні вказівки до лабораторної роботи 1317
Тести 1319
Тести підвищеної складності 1320
Завдання для самостійної роботи 1322
Теми лабораторних робіт 1323
9.2. Ефективні сортування: вихід за межі квадратичної складності 1324
9.2.1 Сортування злиттям 1325
Тести 1332
Тести підвищеної складності 1333

Завдання для самостійної роботи 1336
Теми лабораторних робіт 1336
9.2.2 Швидке сортування 1337
Контрольні запитання 1344
Методичні вказівки до лабораторної роботи 1344
Тести 1346
Тести підвищеної складності 1347
Завдання для самостійної роботи 1350
Теми лабораторних робіт 1351
9.2.3 Пірамідальне сортування 1351
Контрольні запитання 1356
Методичні вказівки до лабораторної роботи 1356
Тести 1358
Тести підвищеної складності 1359
Завдання для самостійної роботи 1362
Теми лабораторних робіт 1362
9.3. Сортування за лінійний час: коли порівняння не потрібні 1363
9.3.1 Сортування підрахунком 1364
Контрольні запитання 1369
Методичні вказівки до лабораторної роботи 1369
Тести 1370
Тести підвищеної складності 1372
Завдання для самостійної роботи 1374
Теми лабораторних робіт 1375
9.4. Гібридні та просунуті сортування: найкраще з усіх світів 1376
9.4.1 Timsort: Прагматичне сортування для реального світу 1377
Контрольні запитання 1381
Методичні вказівки до лабораторної роботи 1381
Тести 1383
Тести підвищеної складності 1384
Завдання для самостійної роботи 1387
Теми лабораторних робіт 1387
9.5. Пошук k-ої порядкової статистики 1388
9.5.1 QuickSelect: Швидкий пошук за лінійний час 1391
Контрольні запитання 1395
Методичні вказівки до лабораторної роботи 1395
Тести 1397
Тести підвищеної складності 1398
Завдання для самостійної роботи 1401
Теми лабораторних робіт 1401

Розділ 10: Алгоритми на рядках та стиснення даних 1402
10.1. Суфіксні структури: суфіксний масив, суфіксне дерево, суфіксний автомат 1403
Контрольні запитання 1417
Методичні вказівки до лабораторної роботи 1417
Тести 1419
Тести підвищеної складності 1420
Завдання для самостійної роботи 1423
Теми лабораторних робіт 1423
10.2. Пошук підрядка 1424
10.2.1 Наївний алгоритм пошуку підрядка 1426
Контрольні запитання 1429
Методичні вказівки до лабораторної роботи 1430

Тести 1431
Тести підвищеної складності 1433
Завдання для самостійної роботи 1435
Теми лабораторних робіт 1436
10.2.2. Алгоритм пошуку підрядка Кнута-Морріса-Пратта (КМП) 1437
Контрольні запитання 1449
Методичні вказівки до лабораторної роботи 1450
Тести 1451
Тести підвищеної складності 1452
Завдання для самостійної роботи 1455
Теми лабораторних робіт 1456
10.1.3. Алгоритм пошуку підрядка Рабіна-Карпа 1456
Контрольні запитання 1462
Методичні вказівки до лабораторної роботи 1462
Тести 1464
Тести підвищеної складності 1465
Завдання для самостійної роботи 1468
Теми лабораторних робіт 1468
10.2.4. Алгоритм пошуку підрядка Боєра-Мура 1469
Модифікація алгоритму Боєра-Мура для пошуку множинних шаблонів 1477
Контрольні запитання 1484
Методичні вказівки до лабораторної роботи 1484
Тести 1486
Тести підвищеної складності 1487
Завдання для самостійної роботи 1490
Теми лабораторних робіт 1490
10.2.6. Алгоритм пошуку підрядка Ахо-Корасік 1491
Контрольні запитання 1500
Методичні вказівки до лабораторної роботи 1500
Тести 1502
Тести підвищеної складності 1503
Завдання для самостійної роботи 1505
Теми лабораторних робіт 1506
10.2.7. Алгоритм пошуку підрядка: Z-алгоритм 1507
Контрольні запитання 1514
Методичні вказівки до лабораторної роботи 1515
Тести 1516
Тести підвищеної складності 1517
Завдання для самостійної роботи 1520
Теми лабораторних робіт 1520
10.3. Редакційна відстань 1521
10.2.1. Алгоритм обчислення відстані Левенштейна 1522
Контрольні запитання 1534
Методичні вказівки до лабораторної роботи 1534
Тести 1536
Тести підвищеної складності 1537
Завдання для самостійної роботи 1539
Теми лабораторних робіт 1540
10.2.2. Алгоритм обчислення відстані Дамерау-Левенштейна 1541
Контрольні запитання 1552
Методичні вказівки до лабораторної роботи 1552
Тести 1554
Тести підвищеної складності 1555

Завдання для самостійної роботи 1557
Теми лабораторних робіт 1558
10.2.3. Алгоритм Гіршберга для обчислення редакційної відстані 1559
Контрольні запитання 1568
Методичні вказівки до лабораторної роботи 1568
Тести 1570
Тести підвищеної складності 1571
Завдання для самостійної роботи 1574
10.4. Алгоритми стиснення 1574
10.4. 1. Перетворення Берроуза-Вілера 1576
Контрольні запитання 1584
Методичні вказівки до лабораторної роботи 1585
Тести 1586
Тести підвищеної складності 1588
Завдання для самостійної роботи 1590
Теми лабораторних робіт 1590
10.4.2 Алгоритми стиснення LZ77 та LZ78 1591
Контрольні запитання 1600
Методичні вказівки до лабораторної роботи 1601
Тести 1603
Тести підвищеної складності 1604
Завдання для самостійної роботи 1606
Теми лабораторних робіт 1606
10.4. Алгоритм кодування Хаффмана 1607
Контрольні запитання 1616
Методичні вказівки до лабораторної роботи 1616
Тести 1618
Тести підвищеної складності 1620
Завдання для самостійної роботи 1622
Теми лабораторних робіт 1622

10.4.4. АЛГОРИТМ ФАНО 1623
Переваги та недоліки 1631
Контрольні запитання 1632
Методичні вказівки до лабораторної роботи 1633
Тести 1634
Тести підвищеної складності 1635
Завдання для самостійної роботи 1638
Теми лабораторних робіт 1639
10.4.4 Алгоритм DEFLATE 1639
Контрольні запитання 1655
Методичні вказівки до лабораторної роботи 1655
Тести 1658
Тести підвищеної складності 1658
Завдання для самостійної роботи 1661
Теми лабораторних робіт 1662

Розділ 11: Обчислювальна геометрія 1663
11.1. Побудова опуклої оболонки 1665
11.1.1. Алгоритм сканування Грема 1666
Контрольні запитання 1675
Методичні вказівки до лабораторної роботи 1676

Тести 1678
Тести підвищеної складності 1679
Завдання для самостійної роботи 1681
Теми лабораторних робіт 1682
11.1.2. Алгоритм Джарвіса 1683
Контрольні запитання 1693
Методичні вказівки до лабораторної роботи 1694
Тести 1696
Тести підвищеної складності 1697
Завдання для самостійної роботи 1699
Теми лабораторних робіт 1700
11.1.3. Алгоритм QuickHull 1701
Контрольні запитання 1711
Методичні вказівки до лабораторної роботи 1712
Тести 1714
Тести підвищеної складності 1715
Завдання для самостійної роботи 1717
Теми лабораторних робіт 1718
11.2. Задачі близькості: пошук найближчої пари точок, діаграма Вороного та тріангуляція Делоне. 1719
11.1.3. Алгоритм пошуку найближчої пари точок 1720
Контрольні запитання 1729
Методичні вказівки до лабораторної роботи 1729
1. Назва 1729
2. Мета 1730
3. Завдання 1730
4. Опис 1730
5. Порядок виконання 1730
6. Результат 1731
Тести 1732
Тести підвищеної складності 1733
Завдання для самостійної роботи 1735
Теми лабораторних робіт 1735
11.3. Перетини та належність: перетин відрізків (Bentley-Ottmann), точка в багатокутнику (Ray casting). 1736
11.3.1. Алгоритм перетину відрізків Бентлі-Оттманна 1739
Контрольні запитання 1752
Методичні вказівки до лабораторної роботи 1753
Тести 1755
Тести підвищеної складності 1756
Завдання для самостійної роботи 1758
Теми лабораторних робіт 1758
11.3.2. Алгоритм променевого трасування 1759
Контрольні запитання 1768
Методичні вказівки до лабораторної роботи 1769
Тести 1771
Тести підвищеної складності 1772
Завдання для самостійної роботи 1774
Теми лабораторних робіт 1775
11.4. Геометричні структури даних: ефективна організація простору 1776

11.4.1. АЛГОРИТМ KD-ДЕРЕВО 1777
Контрольні запитання 1791

Методичні вказівки до лабораторної роботи ... 1791
Тести ... 1793
Тести підвищеної складності ... 1794
Завдання для самостійної роботи ... 1797
Теми лабораторних робіт ... 1798
11.4.1. Алгоритм R-дерево ... 1798
Контрольні запитання ... 1825
Методичні вказівки до лабораторної роботи ... 1826
Тести ... 1828
Тести підвищеної складності ... 1829
Завдання для самостійної роботи ... 1831
Теми лабораторних робіт ... 1832

Розділ 12: Чисельні та криптографічні алгоритми ... 1833

12.1. Теорія чисел ... 1834
12.1.1. Алгоритм Евкліда ... 1834
Контрольні запитання ... 1839
Методичні вказівки до лабораторної роботи ... 1839
Тести ... 1841
Тести підвищеної складності ... 1842
Завдання для самостійної роботи ... 1844
Теми лабораторних робіт ... 1845
12.1.2. Китайська теорема про залишки ... 1846
Контрольні запитання ... 1852
Методичні вказівки до лабораторної роботи ... 1853
Тести ... 1854
Тести підвищеної складності ... 1856
Завдання для самостійної роботи ... 1858
Теми лабораторних робіт ... 1859
Тест Міллера-Рабіна на простоту ... 1860
Контрольні запитання ... 1865
Методичні вказівки до лабораторної роботи ... 1866
Тести ... 1868
Тести підвищеної складності ... 1869
Завдання для самостійної роботи ... 1871
Теми лабораторних робіт ... 1872
Решето Ератосфена ... 1872
Контрольні запитання ... 1879
Методичні вказівки до лабораторної роботи ... 1879
Тести ... 1881
Тести підвищеної складності ... 1882
Завдання для самостійної роботи ... 1885
Теми лабораторних робіт ... 1885
Решето Аткіна ... 1886
Контрольні запитання ... 1893
Методичні вказівки до лабораторної роботи ... 1893
Тести ... 1895
Тести підвищеної складності ... 1896
Завдання для самостійної роботи ... 1898
Теми лабораторних робіт ... 1899

12.2. Криптографія ... 1899

Криптосистема RSA ... 1900
Математичні основи RSA ... 1901
Контрольні запитання ... 1910
Тести ... 1912
Тести підвищеної складності ... 1913
Завдання для самостійної роботи ... 1915
Теми лабораторних робіт ... 1916
Алгоритм Діффі-Геллмана ... 1916
Контрольні запитання ... 1923
Методичні вказівки до лабораторної роботи ... 1923
Тести ... 1925
Тести підвищеної складності ... 1926
Завдання для самостійної роботи ... 1928
Теми лабораторних робіт ... 1929

Криптографія на еліптичних кривих (ECDH, ECDSA) ... 1930
Контрольні запитання ... 1939
Методичні вказівки до лабораторної роботи ... 1939
Тести ... 1941
Тести підвищеної складності ... 1942
Завдання для самостійної роботи ... 1945
Теми лабораторних робіт ... 1945

Криптографічні хеш-функції SHA-2 та SHA-3 ... 1946
Контрольні запитання ... 1957
Методичні вказівки до лабораторної роботи ... 1957
Тести ... 1959
Тести підвищеної складності ... 1960
Завдання для самостійної роботи ... 1962
Теми лабораторних робіт ... 1963

Криптографічна хеш-функція BLAKE2 ... 1963
Контрольні запитання ... 1973
Методичні вказівки до лабораторної роботи ... 1973
Тести ... 1975
Тести підвищеної складності ... 1976
Завдання для самостійної роботи ... 1978
Теми лабораторних робіт ... 1979

Симетричне шифрування AES-GCM ... 1980
Контрольні запитання ... 1987
Методичні вказівки до лабораторної роботи ... 1988
Тести ... 1989
Тести підвищеної складності ... 1990
Завдання для самостійної роботи ... 1993
Теми лабораторних робіт ... 1993

12.3. Алгебраїчні алгоритми ... 1994
Швидке піднесення до степеня ... 1995
Контрольні запитання ... 2002
Методичні вказівки до лабораторної роботи ... 2003
Тести ... 2005
Тести підвищеної складності ... 2006

Завдання для самостійної роботи 2008
Теми лабораторних робіт 2009
Швидке перетворення Фур'є (FFT) 2009
Контрольні запитання 2024
Методичні вказівки до лабораторної роботи 2024
Тести 2026
Тести підвищеної складності 2028
Завдання для самостійної роботи 2030
Теми лабораторних робіт 2030
Алгоритм Штрассена для множення матриць 2031
Контрольні запитання 2039
Методичні вказівки до лабораторної роботи 2040
Тести 2042
Тести підвищеної складності 2043
Завдання для самостійної роботи 2045
Теми лабораторних робіт 2046

ЧАСТИНА VI: ПРОСУНУТІ ТА СУЧАСНІ АЛГОРИТМИ 2047

Розділ 13: Паралельні та розподілені системи 2049
13.1. Паралельні алгоритми: паралельні сортування 2051
Контрольні запитання 2056
Методичні вказівки до лабораторної роботи 2057
Тести 2058
Тести підвищеної складності 2059
Завдання для самостійної роботи 2062
Теми лабораторних робіт 2062
Паралельні сортування з використанням префіксних сум 2063
Контрольні запитання 2069
Методичні вказівки до лабораторної роботи 2070
Тести 2072
Тести підвищеної складності 2073
Завдання для самостійної роботи 2075
Теми лабораторних робіт 2076
13.2. Lock-free/Wait-free структури 2076
Черга Майкла-Скотта 2076
Контрольні запитання 2083
Методичні вказівки до лабораторної роботи 2083
Тести 2085
Тести підвищеної складності 2086
Завдання для самостійної роботи 2088
Теми лабораторних робіт 2089
Стек Трейбера 2089
Контрольні запитання 2096
Методичні вказівки до лабораторної роботи 2097
Тести 2098
Тести підвищеної складності 2099
Завдання для самостійної роботи 2102
Теми лабораторних робіт 2102
13.3. Розподілені алгоритми 2103
Алгоритми консенсусу в розподілених системах 2103
Консенсус Paxos 2105

Контрольні запитання 2112
Методичні вказівки до лабораторної роботи 2113
Тести 2114
Тести підвищеної складності 2115
Завдання для самостійної роботи 2118
Теми лабораторних робіт 2118
Консенсус Raft 2119
Контрольні запитання 2127
Методичні вказівки до лабораторної роботи 2127
Тести 2129

Тести підвищеної складності 2130
Завдання для самостійної роботи 2132
Теми лабораторних робіт 2133

Розподілені хеш-таблиці 2133
Розподілені хеш-таблиці Chord 2134
Контрольні запитання 2142
Методичні вказівки до лабораторної роботи 2143
Тести 2144
Тести підвищеної складності 2145
Завдання для самостійної роботи 2148
Теми лабораторних робіт 2148

РОЗПОДІЛЕНІ ХЕШ-ТАБЛИЦІ KADEMLIA 2149
Контрольні запитання 2159
Методичні вказівки до лабораторної роботи 2160
Тести 2161
Тести підвищеної складності 2162
Завдання для самостійної роботи 2164
Теми лабораторних робіт 2165
13.4. Парадигми обчислень: MapReduce. 2165
Контрольні запитання 2180
Методичні вказівки до лабораторної роботи 2180
Тести 2182
Тести підвищеної складності 2183
Завдання для самостійної роботи 2185
Теми лабораторних робіт 2186

Розділ 14: Алгоритми для великих даних та потоків 2187
14.1. Sketch-структури 2188
Sketch-структури: Count-Min 2189
Контрольні запитання 2201
Методичні вказівки до лабораторної роботи 2201
Тести 2203
Тести підвищеної складності 2204
Завдання для самостійної роботи 2206
Теми лабораторних робіт 2207
Sketch-структури: HyperLogLog 2207
Контрольні запитання 2219
Методичні вказівки до лабораторної роботи 2220
Тести 2221

Тести підвищеної складності 2222
Завдання для самостійної роботи 2225
Теми лабораторних робіт 2225
14.2. Алгоритми для потоків 2225
Lossy Counting 2226
Контрольні запитання 2240
Методичні вказівки до лабораторної роботи 2241
Тести 2243
Тести підвищеної складності 2244
Завдання для самостійної роботи 2246
Теми лабораторних робіт 2246
Space-Saving 2247
Контрольні запитання 2262
Методичні вказівки до лабораторної роботи 2263
Тести 2264
Тести підвищеної складності 2265
Завдання для самостійної роботи 2267
Теми лабораторних робіт 2268
14.3. Структури для зовнішньої пам’яті 2268
LSM-дерева 2269
Контрольні запитання 2284
Методичні вказівки до лабораторної роботи 2285
Тести 2286
Тести підвищеної складності 2287
Завдання для самостійної роботи 2289
Теми лабораторних робіт 2290
Зовнішнє сортування злиттям 2290
Тести 2310
Тести підвищеної складності 2311
Завдання для самостійної роботи 2313
Теми лабораторних робіт 2313
Cache-oblivious алгоритми 2314
Контрольні запитання 2332
Методичні вказівки до лабораторної роботи 2332
Тести 2334
Тести підвищеної складності 2335
Завдання для самостійної роботи 2337
Теми лабораторних робіт 2337

Розділ 15: Розділ 15: Вступ до алгоритмів машинного навчання 2339
Лінійна регресія 2339
Контрольні запитання 2352
Методичні вказівки до лабораторної роботи 2353
Тести 2354
Тести підвищеної складності 2355
Завдання для самостійної роботи 2357
Теми лабораторних робіт 2358
Логістична регресія 2358
Контрольні запитання 2375
Методичні вказівки до лабораторної роботи 2375
Тести 2377
Тести підвищеної складності 2378
Завдання для самостійної роботи 2380

Теми лабораторних робіт 2380
Кластеризація: K-Means 2381
Методи ініціалізації центроїдів 2384
Контрольні запитання 2396
Методичні вказівки до лабораторної роботи 2397
Тести 2398
Тести підвищеної складності 2399
Завдання для самостійної роботи 2401
Теми лабораторних робіт 2402
Кластеризація: DBSCAN 2402
Контрольні запитання 2421
Методичні вказівки до лабораторної роботи 2421
Тести 2423
Тести підвищеної складності 2424
Завдання для самостійної роботи 2426
Теми лабораторних робіт 2426
Ієрархічна кластеризація 2427
Контрольні запитання 2448
Методичні вказівки до лабораторної роботи 2448
Тести 2450
Тести підвищеної складності 2451
Завдання для самостійної роботи 2453
Теми лабораторних робіт 2453
15.3. Класифікація 2454
Класифікація: дерево рішень 2454
Контрольні запитання 2463
Методичні вказівки до лабораторної роботи 2463
Тести 2465
Тести підвищеної складності 2466
Завдання для самостійної роботи 2469
Теми лабораторних робіт 2469
Класифікація: Random Forest 2470
Контрольні запитання 2480
Методичні вказівки до лабораторної роботи 2481
Тести 2483
Тести підвищеної складності 2484
Завдання для самостійної роботи 2486
Теми лабораторних робіт 2487
Класифікація: Gradient Boosting 2487
Контрольні запитання 2498
Методичні вказівки до лабораторної роботи 2498
Тести 2500
Тести підвищеної складності 2501
Завдання для самостійної роботи 2503
Теми лабораторних робіт 2504
Класифікація: SVM 2505
Контрольні запитання 2516
Методичні вказівки до лабораторної роботи 2517
Тести 2518
Тести підвищеної складності 2520
Завдання для самостійної роботи 2522
Теми лабораторних робіт 2522
Наївний байєсівський класифікатор 2523

Контрольні запитання 2534
Методичні вказівки до лабораторної роботи 2534
Тести 2536
Тести підвищеної складності 2537
Завдання для самостійної роботи 2539
Теми лабораторних робіт 2540
15.4. Нейронні мережі 2540
Перцептрон Розенблата 2541
Контрольні запитання 2551
Методичні вказівки до лабораторної роботи 2551
Тести 2553
Тести підвищеної складності 2554
Завдання для самостійної роботи 2556
Теми лабораторних робіт 2557
Багатошарова нейронна мережа та алгоритм зворотного поширення помилки 2557
Контрольні запитання 2571
Методичні вказівки до лабораторної роботи 2572
Тести 2573
Тести підвищеної складності 2575
Завдання для самостійної роботи 2577
Теми лабораторних робіт 2577
Основи згорткових нейронних мереж (CNN) 2578
Контрольні запитання 2589
Методичні вказівки до лабораторної роботи 2590
Тести 2591
Тести підвищеної складності 2592
Завдання для самостійної роботи 2594
Теми лабораторних робіт 2595
Основи рекурентних нейронних мереж (RNN) 2595
Контрольні запитання 2606
Методичні вказівки до лабораторної роботи 2606
Тести 2608
Тести підвищеної складності 2609
Завдання для самостійної роботи 2611
Теми лабораторних робіт 2611
Архітектура трансформера 2612
Контрольні запитання 2623
Методичні вказівки до лабораторної роботи 2624
Тести 2625
Тести підвищеної складності 2626
Завдання для самостійної роботи 2628
Теми лабораторних робіт 2629
Великі мовні моделі (LLM) 2629
Контрольні запитання 2643
Методичні вказівки до лабораторної роботи 2643
Тести 2645
Тести підвищеної складності 2646
Завдання для самостійної роботи 2648
Теми лабораторних робіт 2648

15.5. Зниження розмірності 2649
Метод головних компонент (PCA) 2650
Контрольні запитання 2660

Методичні вказівки до лабораторної роботи ... 2660
Тести ... 2662
Тести підвищеної складності ... 2663
Завдання для самостійної роботи ... 2665
Теми лабораторних робіт ... 2666
Зниження розмірності: t-SNE ... 2666
Контрольні запитання ... 2677
Методичні вказівки до лабораторної роботи ... 2677
Тести ... 2679
Тести підвищеної складності ... 2680
Завдання для самостійної роботи ... 2682
Теми лабораторних робіт ... 2683

Розділ 16: Теорія обчислюваності та складності ... 2683
Машини Тюрінга та нерозв'язні задачі. ... 2684
Обчислюваність і моделі обчислень ... 2685
Задачі для самостійного розв'язання ... 2693
Класи складності ... 2694
Класи складності: P, NP, NP-complete, PSPACE ... 2694
Клас P: Ефективно розв'язувані задачі ... 2695
Клас NP: Задачі з ефективною перевіркою ... 2697
Поліноміальні зведення ... 2699
Клас NP-complete: Найважчі задачі в NP ... 2700
Клас co-NP ... 2701
Клас PSPACE ... 2701
PSPACE-повні задачі ... 2702
Параметризована складність та FPT. ... 2703
Параметризована складність та FPT ... 2703
Онлайн алгоритми та конкурентне відношення: математичний розгляд невизначеності ... 2705
Теоретичні основи: формалізація невизначеності ... 2706
Алгоритм LRU: елегантність простоти ... 2706
Обчислення конкурентних відношень: класичні приклади ... 2707
Методи доведення нижніх меж ... 2708
Рандомізовані онлайн алгоритми ... 2709

Розділ 17: Квантові обчислення чи плутанина? ... 2712
Вступ: Чому квантовий світ "заплутаний"? ... 2712
Криза класичної інтуїції ... 2714
Два стовпи "плутанини": ключові відмінності квантового світу ... 2717
Суперпозиція ... 2718
Фундаментальні поняття ... 2720
Кубіт ... 2720
2. Фізичні основи кубіта ... 2721
2.1. Квантові системи як носії інформації ... 2721
2.2. Математичний опис стану кубіта ... 2722
2.3. Нотація Дірака ... 2722
3. Ключові відмінності кубіта від класичного біта ... 2723
4. Сфера Блоха: Геометрична візуалізація кубіта ... 2724
Чотири стани Белла ... 2727
Створення заплутаності ... 2728
Квантові гейти: Аналоги логічних вентилів для маніпуляції кубітами ... 2728
Математичне представлення квантових гейтів ... 2729
Однокубітні гейти: Фундаментальні операції ... 2729

Фазові гейти 2731
Гейти обертання 2732
Двокубітні гейти: Створення заплутаності 2732
Гейт Тоффолі (CCNOT) 2733
Універсальні набори гейтів 2733
Квантові схеми: Композиція гейтів 2734
Тензорний добуток у квантовій механіці 2734
Проблема опису двох кубітів 2735
Обчислення тензорного добутку векторів 2735
Формування обчислювального базису 2736
Загальний стан двох кубітів 2736
Тензорний добуток довільних станів 2737
Розділені та заплутані стани 2737
Тензорний добуток для трьох і більше кубітів 2738
N кубітів 2739
Операції на тензорних добутках 2739
Часткові виміри та сліди 2740
Квантова телепортація та тензорні добутки 2741
Створення заплутаності 2741
Математичний опис алгоритму 2744
Алгоритм квантової телепортації 2748
Практичний приклад телепортації 2750
Платформи для квантових обчислень 2752
Квантовий паралелізм 2762
Алгоритм Гровера: Квантовий пошук у неструктурованій базі даних 2766
Алгоритм Шора 2773
Алгоритм Шора: Квантова факторизація великих чисел 2773
Математичні основи алгоритму 2773
Класична складність задачі пошуку періоду 2775
Квантове перетворення Фур'є 2779
Тензорна декомпозиція 2780
Складність алгоритму та порівняння з класичними методами 2781
Сучасний стан квантових комп'ютерів 2781
Наслідки для криптографії та інформаційної безпеки 2782
Задачі для самостійного розв'язання 2782
Постквантова криптографія 2783

ГЕНЕТИЧНІ АЛГОРИТМИ 2784

1. Історія розвитку генетичних алгоритмів 2784

2. Класи задач та область застосування генетичних алгоритмів 2786
Задачі комбінаторної оптимізації 2787
Задачі неперервної оптимізації 2789
Багатокритеріальна оптимізація 2790
Задачі машинного навчання 2792
Коли генетичні алгоритми не є оптимальним вибором 2793

3. Біологічна аналогія та головна ідея генетичних алгоритмів 2794
Архітектура життєвого циклу генетичного алгоритму 2797

4. Кодування розв'язків у генетичних алгоритмах 2804

5. Детальний огляд генетичних операторів 2809
Оператори селекції 2809
Оператори кросинговеру (кросоверу) 2815
Оператори мутації 2825
Адаптивні оператори 2831
Функція Растрігіна 2832
Інші генетичні оператори та техніки 2837

ПРИСТОСОВАНІСТЬ У ГЕНЕТИЧНИХ АЛГОРИТМАХ 2840
Біологічна аналогія 2841
Приклади функцій пристосованості 2841
Масштабування пристосованості 2844
Пристосованість та цільова функція 2845
Роль пристосованості в генетичному алгоритмі 2845

Загальний алгоритм генетичного пошуку 2846
Структура базового генетичного алгоритму 2847
Приклад роботи алгоритму 2858
Математичний аналіз збіжності 2860
Налаштування параметрів 2861
Приклади успішного застосування генетичних алгоритмів 2861
Контрольні запитання 2878
Методичні вказівки до лабораторної роботи 2881
Тести (базовий рівень) 2886
Тести підвищеної складності 2889
Завдання для самостійної роботи 2899
Теоретичні завдання 2899
Дослідницькі завдання (підвищеної складності) 2901
Теми лабораторних робіт 2902
Базові лабораторні роботи (обов'язкові) 2902
Спеціалізовані лабораторні роботи (за вибором) 2905

Список використаної літератури. 2909
Додаток 1 2949

Частина I: Фундаментальні концепції та лінійні структури

Розділ 1: Вступ та аналіз алгоритмів

1.1. Поняття алгоритму, структури даних та абстрактного типу даних (ADT).

Ласкаво просимо до світу комп’ютерних наук! Перш ніж ми почнемо будувати складні програмні системи, нам потрібно закласти міцний фундамент. Цей фундамент стоїть на трьох китах: алгоритмах, структурах даних та абстрактних типах даних. Розуміння цих концепцій та зв’язку між ними — це те, що відрізняє простого кодера від професійного інженера програмного забезпечення.

У цьому розділі ми детально розглянемо кожну з цих концепцій, щоб ви не просто знали визначення, а й глибоко розуміли їхню суть та роль у створенні ефективного та надійного програмного забезпечення.

1. Поняття алгоритму

1.1. Історія виникнення та суть поняття

Ідея алгоритму є значно старшою за комп’ютери. Сама назва «алгоритм» походить від імені видатного перського математика IX століття Мухаммада ібн Муси аль-Хорезмі. У своїй праці «Про індійський рахунок» він описав правила виконання арифметичних операцій в десятковій системі числення. Латинізована версія його імені, *Algorithmi*, і дала назву самому поняттю.

По суті, аль-Хорезмі зробив революційну річ: він формалізував набір чітких, послідовних інструкцій, які гарантовано призводять до правильного результату (наприклад, при додаванні чи множенні чисел). Це і є серцевиною будь-якого алгоритму — чіткий план дій для розв’язання задачі.

Протягом століть математики розробляли нові алгоритми. Наприклад, алгоритм Евкліда для знаходження найбільшого спільного дільника, створений ще в Давній Греції, є одним з найстаріших і досі використовуваних алгоритмів.

Сучасне, більш формальне розуміння алгоритму почало формуватися в XIX та XX століттях з розвитком математичної логіки та теорії обчислень.

- Чарльз Беббідж та Ада Лавлейс у XIX столітті, працюючи над проєктом «Аналітичної машини», по суті, розробили перші складні програми,

що були алгоритмами, призначеними для механічного виконання. Аду Лавлейс часто називають першою програмісткою саме за її детальний опис алгоритму обчислення чисел Бернуллі для машини Беббіджа.

– Алан Тюрінг у 1936 році надав суворе математичне визначення алгоритму через свою концепцію «машини Тюрінга» — абстрактного обчислювального пристрою. Він довів, що існують задачі, які неможливо розв'язати алгоритмічно (наприклад, проблема зупинки).
– Робота над розшифровкою німецького коду «Енігма» під час Другої світової війни також стала потужним поштовхом для розвитку практичних алгоритмів та створення перших обчислювальних машин.

Сьогодні алгоритми є скрізь: від приготування кави у кавомашині та роботи GPS-навігатора до торгівлі на фондовій біржі та аналізу геному людини.

1.2. Визначення та властивості алгоритму

Хоча інтуїтивне розуміння алгоритму як «рецепту» або «інструкції» є корисним, в інформатиці використовується більш строге визначення.

Алгоритм — це скінченна послідовність чітко визначених, однозначних інструкцій, призначених для розв'язання певного класу задач, яка приводить до отримання результату за скінченну кількість кроків.

Щоб послідовність інструкцій могла називатися алгоритмом, вона повинна мати наступні властивості:

1. Дискретність. Алгоритм складається з окремих, послідовних кроків (інструкцій). Неможливо виконати «півтора кроку». Процес виконання розбивається на чіткі етапи.
2. Детермінованість (однозначність). Кожна інструкція алгоритму має бути абсолютно точною і не допускати двозначних тлумачень. Виконавець, отримавши однакові вхідні дані, має щоразу виконувати однакову послідовність дій і отримувати однаковий результат. *(Примітка: існують також імовірнісні алгоритми, але класичне визначення вимагає детермінізму).*
3. Скінченність (завершуваність). Алгоритм має гарантовано завершити свою роботу за скінченну кількість кроків для будь-яких допустимих вхідних даних. Алгоритм, що може зациклитися, не є коректним.

4. Результативність (коректність). По завершенні роботи алгоритм має видавати правильний результат для поставленої задачі. Він повинен робити саме те, для чого був створений.
5. Масовість. Алгоритм розробляється для розв'язання не однієї конкретної задачі, а цілого класу однотипних задач. Наприклад, алгоритм сортування має працювати для будь-якого масиву чисел, а не лише для масиву `{3, 1, 2}`.

1.3. Способи опису алгоритмів

Існує декілька способів представити алгоритм:

- Словесний опис: Використання звичайної мови. Зручно для загального пояснення ідеї, але може бути неточним і громіздким.
- Блок-схеми: Графічне представлення з використанням стандартизованих блоків для різних операцій (початок/кінець, введення/виведення, обчислення, умова). Наочно, але незручно для складних алгоритмів.
- Мови програмування: Опис алгоритму конкретною мовою (C++, Python, Java). Це фінальний етап, але такий опис є надто детальним і залежним від синтаксису мови.
- Псевдокод: Найкращий варіант для вивчення та проєктування. Це система позначень, що використовує синтаксичні конструкції мов програмування, але є більш абстрактною та не прив'язаною до конкретної мови. Саме псевдокод ми будемо використовувати в цьому підручнику.

Приклад: Псевдокод для пошуку максимального елемента в масиві

```
ФУНКЦІЯ FindMax(Array A):
    // Припускаємо, що перший елемент є максимальним
    max_element = A[0]

    // Проходимо по решті елементів масиву
    ДЛЯ і ВІД 1 ДО довжина(A)-1:
        // Якщо знаходимо елемент, більший за поточний максимум,
        // оновлюємо максимум.
        ЯКЩО A[i] > max_element ТО
            max_element = A[i]
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    // Повертаємо знайдений максимум
```

```
    ПОВЕРНУТИ max_element
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Класичний приклад: алгоритм Евкліда

Одним з найперших та найелегантніших алгоритмів, відомих людству, є алгоритм Евкліда для знаходження найбільшого спільного дільника (НСД) двох цілих чисел.

Постановка задачі: Для двох заданих цілих чисел `a` та `b` знайти найбільше ціле число, на яке `a` і `b` діляться без остачі. Наївний підхід полягав би в тому, щоб перебирати всі числа від меншого з `a` і `b` до 1 і перевіряти, чи ділить воно обидва числа. Це дуже неефективно для великих чисел.

Ідея Евкліда: Фундаментальна ідея базується на наступному спостереженні: Найбільший спільний дільник двох чисел не зміниться, якщо більше число замінити на різницю між більшим і меншим числом. Наприклад, `НСД(48, 18) = НСД(48-18, 18) = НСД(30, 18)`. Повторюючи цей процес, ми зрештою прийдемо до того, що одне з чисел стане 0, і тоді друге число і буде НСД.

Більш ефективна версія цього ж принципу використовує остачу від ділення: НСД(a, b) = НСД$(b, a \pmod b)$, де $a \pmod b$ — це остача від ділення `a` на `b`.

Давайте прослідкуємо цей процес на прикладі `НСД(48, 18)`:

6. `a=48, b=18`. 48 (mod 18) = 12. Тепер шукаємо `НСД(18, 12)`.
7. `a=18, b=12`. 18 (mod 12) = 6. Тепер шукаємо `НСД(12, 6)`.
8. `a=12, b=6`. $12 = 0`. Ми дійшли до остачі 0.
9. Процес завершено. Останній ненульовий залишок (6) і є найбільшим спільним дільником.

Псевдокод алгоритму Евкліда (рекурсивна версія): Ця версія є прямою реалізацією математичного співвідношення.

```
ФУНКЦІЯ GCD_Recursive(a, b):
    // Базовий випадок рекурсії: якщо b дорівнює 0, то a є НСД.
    ЯКЩО b == 0 ТО
        ПОВЕРНУТИ a
    КІНЕЦЬ ЯКЩО

    // Рекурсивний крок: викликаємо функцію для b та залишку від діле
ння a на b.
    ПОВЕРНУТИ GCD_Recursive(b, a % b)
КІНЕЦЬ ФУНКЦІЇ
```

Псевдокод алгоритму Евкліда (ітеративна версія): Ця версія є більш ефективною на практиці, оскільки не використовує рекурсивних викликів.

```
ФУНКЦІЯ GCD_Iterative(a, b):
    // Поки b не дорівнює 0, ми продовжуємо цикл.
    ПОКИ b != 0:
        temp = b
        b = a % b
        a = temp
    КІНЕЦЬ ПОКИ
    ПОВЕРНУТИ a
КІНЕЦЬ ФУНКЦІЇ
```

Алгоритм Евкліда є надзвичайно ефективним. Його складність є логарифмічною відносно меншого з чисел, $O(\log(\min(a,b)))$, що дозволяє знаходити НСД для дуже великих чисел практично миттєво.

1.5. Математичний аналіз алгоритмів і асимптотична нотація

Щоб порівнювати алгоритми та робити усвідомлений вибір, потрібен формальний інструмент для вимірювання їхньої ефективності. Ним є аналіз складності алгоритмів. Ми оцінюємо:

- Часову складність — скільки елементарних операцій виконує алгоритм залежно від розміру вводу n.
- Просторову складність — скільки пам'яті використовує алгоритм.

Найуживаніші позначення:

- $O(g(n))$ (Big-O) — верхня межа: алгоритм виконує не більше `c·g(n)` операцій для достатньо великих n.
- $\Omega(g(n))$ (Omega) — нижня межа.
- $\Theta(g(n))$ (Theta) — точна асимптота, коли виконуються обидві попередні нерівності.

Приклад. Сортування вставками має $O(n^2)$ у найгіршому випадку й $\Theta(n)$ у найкращому (вже відсортований масив).

Також розрізняють середній та найгірший випадки; останній важливий для критичних систем — наприклад, у реальному часі.

1.6. Класифікація алгоритмів за парадигмами

1. Divide & Conquer (Поділяй і володарюй) — рекурсивний поділ задачі на підзадачі. *Приклади: MergeSort, QuickSort, Karatsuba, FFT.*
2. Greedy (Жадібні) — локально оптимальний крок → глобальне рішення. *Приклади: алгоритм Крускала, Prim, Dijkstra (без від'ємних ребер).*

3. Dynamic Programming (Динамічне) — перекриття підзадач + збереження проміжних результатів. *Приклади: Floyd–Warshall, LIS, Knapsack, Bellman-Ford.*
4. Backtracking & Branch-and-Bound — повний перебір із «відкатом» і відсіканням.
5. Randomised / Monte-Carlo / Las-Vegas — випадкові вибори, статистичні гарантії.
6. Паралельні та розподілені — MapReduce, GPU-kernel, PRAM-моделі.
7. Approximation & Meta-heuristics — GA, Simulated Annealing, Tabu Search.

1.7. Життєвий цикл алгоритму — від ідеї до оптимізації

Розробка хорошого алгоритму — це не одномоментний акт натхнення, а структурований інженерний процес, що складається з кількох ключових етапів. Розуміння цього життєвого циклу дозволяє підходити до розв'язання задач системно та професійно.

8. Формалізація задачі. Це найважливіший перший крок. Перш ніж писати код, потрібно чітко зрозуміти, яку саме задачу ми розв'язуємо. На цьому етапі ми повинні визначити:
    - Вхідні дані: Які дані отримує алгоритм? Який їхній тип, формат та обсяг?
    - Вихідні дані: Що саме має бути результатом роботи алгоритму?
    - Обмеження: Які є обмеження на вхідні дані (наприклад, числа додатні, масив не порожній)? Які є обмеження на ресурси (час виконання, пам'ять)? Помилка на цьому етапі, як і при будівництві, може призвести до того, що ми збудуємо міцний, але абсолютно непотрібний міст не в тому місці.
9. Вибір абстрактного типу даних (АТД). На ранньому етапі проєктування ми обираємо логічну модель для наших даних. Чи нам потрібна проста послідовність (АТД «Список»), чи колекція унікальних елементів (АТД «Множина»), чи, можливо, пари ключ-значення (АТД «Словник»)? Цей вибір фундаментально впливає на те, які алгоритми ми зможемо застосувати.
10. Дизайн та ескіз алгоритму. На цьому етапі ми створюємо високорівневий «креслюнок» нашого розв'язку. Ми ще не думаємо про конкретну мову програмування. Головний інструмент тут —

псевдокод, який дозволяє описати логіку алгоритму, використовуючи конструкції, зрозумілі будь-якому програмісту. Також можуть використовуватися блок-схеми для візуалізації потоку виконання.

11. Доведення коректності. Як переконатися, що наш алгоритм завжди буде працювати правильно? Для цього існують формальні методи. Наприклад, для доведення коректності циклів використовують інваріанти циклу — твердження, яке є істинним перед першою ітерацією і залишається істинним після кожної наступної. Для рекурсивних алгоритмів застосовують метод математичної індукції.

12. Асимптотичний аналіз. Тут ми аналізуємо, як поводитиметься наш алгоритм при збільшенні обсягу вхідних даних. Ми оцінюємо його часову та просторову складність за допомогою асимптотичних нотацій (O, Θ, Ω). Це дозволяє нам порівнювати різні алгоритми на теоретичному рівні та відкидати завідомо неефективні.

13. Реалізація та оптимізація. Після того, як ми впевнились у коректності та асимптотичній ефективності алгоритму, ми пишемо код на конкретній мові програмування. Далі може слідувати етап низькорівневої оптимізації, де ми враховуємо особливості архітектури сучасних процесорів:

    - Кеш-ефективність (Cache-friendliness): Проєктування доступу до даних так, щоб максимально використовувати швидкі кеші процесора.
    - Паралелізм: Адаптація алгоритму для виконання на багатоядерних системах.
    - Векторизація: Використання спеціалізованих інструкцій процесора (SIMD) для одночасної обробки кількох елементів даних.

14. Емпіричне тестування та профілювання. Теорія — це добре, але практика — критерій істини. На цьому фінальному етапі ми «проганяємо» нашу реалізацію на широкому наборі тестових даних, вимірюємо реальний час виконання та використання пам’яті за допомогою спеціальних інструментів — профайлерів. Це дозволяє знайти «гарячі точки» в коді та переконатися, що наш алгоритм ефективний не лише в теорії, але й на практиці.

2. Структури даних

2.1. Визначення та потреба

Якщо алгоритм — це рецепт, то структура даних — це те, як ви організовуєте ваші інгредієнти на кухні. Ви можете скласти все в одну велику купу (погано), розкласти по підписаних банках на полицях (краще) або зберігати найпотрібніші спеції в маленьких ємностях біля плити (ще краще). Ваш вибір організації інгредієнтів (даних) безпосередньо впливає на те, наскільки швидко та зручно ви (алгоритм) зможете готувати.

Структура даних — це спосіб організації, зберігання та управління даними у комп'ютерній пам'яті, який дозволяє ефективно виконувати над ними певні операції.

Навіщо вони потрібні? Програма рідко працює з одним-єдиним числом. Зазвичай ми маємо справу з величезними колекціями даних. Структури даних дозволяють нам:

- Ефективно зберігати великі обсяги інформації.
- Швидко отримувати доступ до потрібних даних.
- Легко додавати, видаляти та модифікувати дані.

Вибір правильної структури даних є одним з найважливіших рішень при проєктуванні програми. Неправильний вибір може зробити навіть найкращий алгоритм катастрофічно повільним.

2.2. Класифікація структур даних

Структури даних можна класифікувати за різними критеріями.

За лінійністю:

- Лінійні: Елементи розташовані послідовно, один за одним. Кожен елемент (окрім першого та останнього) має одного попередника та одного наступника.
- Нелінійні: Елементи мають складніші, ієрархічні або мережеві зв'язки. Один елемент може мати декілька наступників або попередників.

За статичністю:

- Статичні: Розмір структури фіксується під час її створення і не може бути змінений.
- Динамічні: Розмір структури може змінюватися під час виконання програми.

2.3. Детальний огляд основних структур даних

Масив (Array)

- Ідея: Найпростіша структура даних. Уявіть собі довгий ряд комірок для зберігання, кожна з яких має свій унікальний номер (індекс), починаючи з 0.
- Представлення: Суцільний блок пам'яті, де елементи зберігаються один за одним.
- Ключові операції та складність:
  - Доступ до елемента за індексом (`A[i]`): $O(1)$. Це відбувається миттєво, оскільки комп'ютер може математично обчислити точну адресу елемента.
  - Пошук елемента за значенням: $O(n)$. Потрібно переглянути всі елементи.
  - Вставка/видалення в середині: $O(n)$. Потрібно «зсунути» всі наступні елементи.
- Коли використовувати: Коли вам потрібен дуже швидкий доступ до даних за їхнім порядковим номером і кількість елементів не буде часто змінюватися.

Зв'язаний список (Linked List)

- Ідея: Гнучкий ланцюжок, де кожен елемент (вузол) не тільки зберігає дані, але й «знає», де знаходиться наступний елемент.
- Представлення: Набір об'єктів-вузлів, розкиданих по пам'яті. Кожен вузол містить дані та вказівник на наступний вузол. Існують також двозв'язні списки, де вузол має вказівники і на наступний, і на попередній елемент.
- Ключові операції та складність:
  - Доступ до елемента за індексом: $O(n)$. Щоб дістатися до 100-го елемента, потрібно пройти 99 попередніх.
  - Пошук елемента за значенням: $O(n)$.
  - Вставка/видалення на початку: $O(1)$.
  - Вставка/видалення в довільному місці (якщо є вказівник на сусідній вузол): $O(1)$.

– Коли використовувати: Коли вам потрібно часто і швидко додавати або видаляти елементи з довільних місць у послідовності, а швидкий доступ за індексом не є пріоритетом.

Стек (Stack)

– Ідея: Стопка тарілок або книг. Ви можете покласти нову тарілку тільки зверху і взяти можете теж тільки верхню. Цей принцип називається LIFO (Last-In, First-Out) — «останнім прийшов, першим пішов».

– Представлення: Це абстрактний тип даних, який може бути реалізований на основі масиву або зв'язаного списку.

– Ключові операції та складність:

- **push** (додати на вершину): $O(1)$.
- **pop** (видалити з вершини): $O(1)$.

– Коли використовувати: Для управління викликами функцій (call stack), реалізації функції «Undo», перевірки дужкових послідовностей.

Черга (Queue)

– Ідея: Звичайна черга людей в магазині. Нові люди стають в кінець, а обслуговують того, хто стоїть на початку. Цей принцип називається FIFO (First-In, First-Out) — «першим прийшов, першим пішов».

– Представлення: Також абстрактний тип даних, реалізується на масиві (як кільцевий буфер) або зв'язаному списку.

– Ключові операції та складність:

- **enqueue** (додати в кінець): $O(1)$.
- **dequeue** (видалити з початку): $O(1)$.

– Коли використовувати: Для управління задачами, що виконуються по черзі, в мережевих буферах, для алгоритму пошуку в ширину (BFS).

Хеш-таблиця (Hash Table)

– Ідея: «Магічна» картотека. Ви даєте їй ключ (наприклад, прізвище студента), спеціальна функція (хеш-функція) миттєво обчислює номер ящика, де зберігається інформація.

– Представлення: Масив, поєднаний з хеш-функцією, яка перетворює ключ на індекс масиву.

– Ключові операції та складність:

- Вставка, видалення, пошук за ключем: $O(1)$ в середньому. У найгіршому випадку (коли багато різних ключів потрапляють в один «ящик») складність може впасти до $O(n)$.

– Коли використовувати: Коли потрібен найшвидший можливий пошук, вставка та видалення за унікальним ключем. Це основа для словників (`map`), множин (`set`) та багатьох видів кешу.

Дерево (Tree)

– Ідея: Ієрархічна структура, що нагадує родинне дерево або файлову систему. Є один головний елемент (корінь), від якого відходять «гілки» до нащадків.

– Представлення: Вузлова структура, де кожен вузол містить дані та вказівники на своїх нащадків.

– Ключові операції та складність: Залежать від типу дерева. Для збалансованих бінарних дерев пошуку (BST) пошук, вставка та видалення виконуються за $O(\log n)$.

– Коли використовувати: Для представлення ієрархій, ефективного пошуку та сортування даних (BST), реалізації черг з пріоритетами (купи).

Граф (Graph)

– Ідея: Модель мережі, що складається з вершин (об’єктів) та ребер (зв’язків між ними).

– Представлення: Зазвичай у вигляді матриці суміжності або списків суміжності.

– Ключові операції та складність: Залежать від конкретного алгоритму (пошук шляху, обхід тощо).

– Коли використовувати: Для моделювання будь-яких мережевих структур: соціальних мереж, доріг, комп’ютерних мереж, залежностей між задачами.

2.4. Як вибір структури даних впливає на швидкодію

Це ключове питання. Розглянемо простий приклад: нам потрібно зберігати список студентів і часто виконувати дві операції: додавати нового студента та шукати студента за прізвищем.

– Варіант 1: Динамічний масив (вектор).

- *Додавання:* Дуже швидко, $O(1)$ в середньому.

- *Пошук:* Нам доведеться переглядати всіх студентів по черзі, доки не знайдемо потрібного. У найгіршому випадку — $O(n)$. Якщо студентів мільйон, це дуже повільно.

– Варіант 2: Відсортований динамічний масив.

- *Додавання:* Щоб зберегти порядок, нам доведеться «розсунути» масив, щоб вставити нового студента. Це повільно — $O(n)$.
- *Пошук:* Оскільки масив відсортований, ми можемо використовувати ефективний бінарний пошук. Складність — $O(\log n)$. Для мільйона студентів це буде надзвичайно швидко.

– Варіант 3: Хеш-таблиця (ключ — прізвище).

- *Додавання:* Дуже швидко, $O(1)$ в середньому.
- *Пошук:* Дуже швидко, $O(1)$ в середньому.

Висновок: Для нашої задачі хеш-таблиця є найкращим вибором. Якби нам потрібно було часто виводити список студентів за алфавітом, можливо, краще підійшло б збалансоване бінарне дерево пошуку (яке ми розглянемо пізніше).

Не існує «найкращої» структури даних на всі випадки життя. Вибір завжди є компромісом між швидкістю виконання різних операцій та обсягом пам’яті, що використовується. Завдання інженера — проаналізувати вимоги задачі та обрати найоптимальнішу структуру.

2.5. Розширений каталог структур даних та їх асимптотики

Клас	Структури	Пам’ять	Доступ	Пошук	Вставка	Видалення
Купи	Binary-Heap, Fibonacci-Heap, Pairing-Heap	$O(n)$	—	—	$O(\log n)$ / *F-heap*: $O(1)$ amort.	$O(\log n)$
Хеш-структури	Cuckoo-Hash, Robin-Hood, Hopscotch	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Дерева пошуку	AVL, Red-Black, B-Tree	$O(n)$	$O(\log n$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Клас	Структури	Пам’ять	Доступ	Пошук	Вставка	Видалення
Пробабілістичні	Skip List, Bloom Filter	$O(n)$ / $O(m)$	$O(\log n$	BF: помилк.	$O(\log n)$	$O(\log n)$ / BF: неможл.
Гео-просторові	Quad-, R-, KD-Tree	$O(n)$	$O(\log n$	$O(\log n)$	$O(\log n)$	$O(\log n)$

m — кількість бітів у Bloom-фільтрі.

2.5. Спеціалізовані структури даних для зовнішньої пам’яті

Коли обсяг даних настільки великий, що не вміщується в оперативну пам’ять (RAM), правила гри змінюються. Доступ до жорсткого диска або SSD є на порядки повільнішим за доступ до RAM. Для таких випадків були розроблені спеціальні структури даних.

2.5.1. LSM-Дерева

Проблема: Традиційні збалансовані дерева (як B-дерева), що використовуються в базах даних, є ефективними для читання, але страждають від повільних випадкових записів. Кожна вставка чи оновлення може вимагати зчитування кількох блоків з диска, їх модифікації та запису назад, що є дуже повільною операцією.

Ідея LSM-дерева: Перетворити багато повільних, випадкових записів на один швидкий, послідовний запис. Аналогія: замість того, щоб щоразу ставити нову книгу на правильне місце у величезній бібліотеці (повільно), ми спершу складаємо нові книги у невеликий ящик біля входу (швидко). І лише коли ящик заповнюється, ми беремо всі книги з нього, сортуємо їх і розставляємо по полицях бібліотеки за один раз.

Дерева складаються з таких компонент:

1. MemTable: Невелика структура даних в оперативній пам’яті (зазвичай збалансоване дерево), яка приймає всі нові операції запису. Запис в RAM — це дуже швидко.

2. SSTable: Коли MemTable заповнюється, всі дані з неї сортуються і записуються на диск як один незмінний файл. Цей запис є послідовним, а отже, дуже швидким.

3. Компактифікація: У фоновому режимі періодично запускається процес, який зливає декілька SSTable-файлів в один більший, видаляючи застарілі або видалені дані.

Профіль продуктивності:

Запис: Надзвичайно швидкий ($O(1)$, оскільки йде в пам'ять).

Читання: Потенційно повільне. Щоб знайти дані, потрібно спершу перевірити MemTable, а потім послідовно декілька SSTable-файлів на диску. Для прискорення читання використовуються фільтри Блума.

LSM-дерева ідеально підходять для систем з великою кількістю операцій запису, як-от бази даних Cassandra, LevelDB, RocksDB.

2.5.2. Персистентні (Функціональні) структури даних

Проблема: У класичному (імперативному) програмуванні при зміні структури даних її попередня версія знищується. Наприклад, додавши елемент в масив, ви вже не можете легко повернутися до стану масиву «до додавання».

Ідея персистентності: Жодних змін! Замість того, щоб модифікувати існуючу структуру, при будь-якій операції ми створюємо її нову версію. При цьому стара версія залишається недоторканою і доступною. Це досягається завдяки розумному спільному використанню пам'яті між версіями.

Основний прийом — копіювання шляху: Уявіть бінарне дерево. Якщо нам потрібно змінити один листок, ми не копіюємо все дерево. Ми створюємо новий листок. Потім ми створюємо нову копію його батька, яка вказує на новий листок (і на старого «брата»). Потім ми створюємо копію дідуся, і так далі, аж до кореня. Всі гілки, яких зміна не торкнулася, просто використовуються за посиланням. В результаті ми отримуємо два корені — для старої та нової версії дерева, які ділять між собою більшу частину вузлів.

Переваги:

Безпека у багатопотоковому середовищі: Оскільки дані ніколи не змінюються, читачам не потрібні блокування.

Історія версій та функція «Undo»: Дуже легко реалізувати відкат до попереднього стану — достатньо просто зберегти посилання на старий корінь.

Персистентні структури є основою функціональних мов програмування (Haskell, Clojure, F#).

2.6. Архітектурні аспекти: кеші та локальність посилань

Сучасні процесори значно швидші за оперативну пам'ять. Щоб уникнути простоїв, вони використовують ієрархію швидких кешів (L1, L2, L3). Алгоритм, що не враховує цього, може працювати на порядки повільніше, ніж міг би.

Локальність посилань (Locality of Reference) — це принцип, згідно з яким процесор найближчим часом, ймовірно, звернеться до даних, що знаходяться поруч з тими, до яких він щойно звертався.

– Просторова локальність: Якщо ви звернулись до елемента масиву `A[i]`, велика ймовірність, що наступним вам знадобиться `A[i+1]`. Коли процесор завантажує `A[i]` з RAM, він зазвичай завантажує цілу «кеш-лінію» (наприклад, 64 байти), що містить і сусідні елементи. Масиви мають чудову просторову локальність. Зв'язані списки, навпаки, є жахом для кешу: їхні вузли розкидані по всій пам'яті, і кожен перехід по вказівнику може призводити до дорогого промаху кешу (cache miss).

– Часова локальність: Якщо ви звернулись до даних, велика ймовірність, що ви звернетесь до них знову найближчим часом.

Проектуючи алгоритми, потрібно прагнути до лінійного доступу до даних. Алгоритми, що оптимізують доступ до пам'яті без явного знання розміру кешу, називаються cache-oblivious.

2.7. Перевірені шаблони вибору структури даних

Вибір структури даних — це завжди пошук компромісу. Ось декілька перевірених часом шаблонів:

1. Навантаження: багато читання, мало запису ($R \gg W$). Якщо дані рідко змінюються, але до них часто звертаються, головне — швидкий пошук.
    - Рішення: Відсортовані масиви (для бінарного пошуку), збалансовані дерева (B+-дерева в базах даних), хеш-таблиці.
2. Навантаження: багато запису, мало читання ($W \gg R$). Якщо система постійно приймає нові дані, головне — швидка вставка.
    - Рішення: Журнальні структури, такі як LSM-дерева, черги повідомлень, стеки.
3. Потреба: швидкий пошук за ключем.

- Рішення: Хеш-таблиця — вибір за замовчуванням. Середній час операцій — $O(1)$.

4. Потреба: запити по діапазону («знайти всіх користувачів віком від 20 до 30»).
    - Рішення: Хеш-таблиці тут безсилі. Потрібні впорядковані структури: збалансовані дерева пошуку (AVL, Red-Black) або спеціалізовані дерева відрізків (Segment Tree).
5. Потреба: аналітика потокових даних («порахувати кількість унікальних відвідувачів сайту за добу»).
    - Рішення: Зберігати всіх відвідувачів дорого. Використовують імовірнісні структури даних, як-от HyperLogLog, які дають наближену відповідь з неймовірною ефективністю по пам’яті.

3. Абстрактний тип даних (АТД): поглиблений погляд

3.1. Ідея абстракції: «Що» проти «Як»

Ми підійшли до останньої, найбільш абстрактної, але вкрай важливої концепції. Уявіть, що ви сідаєте за кермо автомобіля. У вас є кермо, педалі, важіль перемикання передач. Це ваш інтерфейс для взаємодії з автомобілем. Ви знаєте, що станеться, якщо ви натиснете на гальма (машина зупиниться), але вам абсолютно не потрібно знати, як саме це відбувається: чи у вас дискові гальма, чи барабанні, як працює гідравлічна система. Виробник автомобіля надав вам чіткий контракт: «натискання на цю педаль призводить до гальмування».

Абстрактний тип даних (АТД) — це такий самий «інтерфейс» або «контракт» у програмуванні. Це спосіб мислення про дані на логічному рівні, повністю ігноруючи деталі їх фізичного зберігання в пам’яті.

3.2. Формальне визначення та компоненти АТД

Абстрактний тип даних — це математична модель, що складається з двох частин:

1. Множина значень: Опис того, які дані може зберігати АТД (наприклад, «колекція цілих чисел», «набір унікальних рядків»).
2. Набір операцій (Interface): Чіткий перелік операцій, які можна виконувати над цими даними. Це публічна частина «контракту».

Наприклад, для АТД «Стек» це виглядає так:

– Множина значень: Впорядкована колекція елементів.

– Набір операцій:
    - `push(item)`: додати елемент на вершину стека.
    - `pop()`: видалити та повернути елемент з вершини стека.
    - `peek()`: подивитись на елемент на вершині, не видаляючи його.
    - `isEmpty()`: перевірити, чи стек порожній.

Але цього недостатньо. Щоб контракт був повним, потрібна третя частина:

3. Аксіоми (Axioms/Semantics): Набір правил, що описують поведінку операцій та їх взаємодію. Аксіоми гарантують, що будь-яка реалізація АТД буде поводитися очікувано.
    - *Приклад аксіом для АТД «Стек»:*
        - `isEmpty(newStack())` -> `true` (новий стек завжди порожній).
        - `isEmpty(push(S, x))` -> `false` (стек після додавання елемента не може бути порожнім).
        - `pop(push(S, x))` -> `S` (якщо додати елемент, а потім видалити, стек повернеться до початкового стану).
        - `peek(push(S, x))` -> `x` (елемент на вершині — це останній доданий елемент).

Саме ці три компоненти — значення, операції та аксіоми — повністю визначають абстрактний тип даних.

3.3. АТД проти Структура Даних: у чому різниця?

Це фундаментальна відмінність, яку потрібно зрозуміти раз і назавжди.

– АТД — це логічний опис, концепція, контракт, «що» можна робити.

– Структура даних — це конкретна фізична реалізація цього контракту в пам’яті, «як» це зроблено.

Давайте розглянемо інший приклад — АТД «Список» (List).

Контракт АТД «Список»:

– Дані: Впорядкована колекція елементів.

– Операції: `add(index, item)`, `remove(index)`, `get(index)`, `size()`.

Можливі реалізації (структури даних):

1. На основі динамічного масиву:

- *Як це працює*: Використовується суцільний блок пам'яті.
- *Профіль продуктивності:* `get(index)` - $O(1)$; `add/remove` в кінці - $O(1)$; `add/remove` в середині - $O(n)$.

2. На основі двозв'язного списку:
    - *Як це працює:* Ланцюжок вузлів, де кожен «знає» про сусіда зліва та справа.
    - *Профіль продуктивності*: `get(index)` - $O(n)$; `add/remove` на початку/в кінці - $O(1)$; `add/remove` в середині (якщо є ітератор) - $O(1)$.

Обидві реалізації повністю задовольняють контракту АТД «Список», але мають кардинально різний профіль продуктивності.

3.4. Сила абстракції: навіщо потрібні АТД?

Використання АТД є основою сучасного об'єктно-орієнтованого та модульного програмування.

– Абстракція та Інкапсуляція: Це дозволяє нам проєктувати складні системи, не занурюючись у низькорівневі деталі. Ми приховуємо складність реалізації від «клієнта» (іншої частини програми, що використовує наш код). Клієнту не потрібно знати, як працює наш список всередині, йому достатньо знати про методи `add` та `get`.

– Модульність та гнучкість: Це найважливіша практична перевага. Ми можемо легко замінити одну реалізацію на іншу, не змінюючи код, який її використовує.
    - *Приклад:* Ми написали програму, використовуючи список на основі масиву. Після тестування ми виявили, що найчастіша операція — це видалення елементів з початку списку, що працює дуже повільно. Ми можемо зайти в наш модуль списку і замінити реалізацію на двозв'язний список. Весь інший код програми залишиться незмінним, але продуктивність значно зросте.

– Коректність: Формальний опис АТД через аксіоми дозволяє математично доводити коректність алгоритмів, що використовують цей АТД, незалежно від майбутніх змін у його реалізації.

3.5. Дизайн за контрактом (Design by Contract) та формальна верифікація АТД

«Дизайн за контрактом» — це методологія проєктування ПЗ, яка розглядає взаємодію між компонентами системи як формальний контракт. Для АТД це означає чітке визначення «обов'язків» та «прав» як для реалізації структури, так і для коду, що її використовує.

- Передумови (Preconditions): Умови, які мають бути істинними *перед* викликом операції. Це обов'язок «клієнта». Наприклад, для операції `pop()` зі стека передумовою є те, що стек не порожній.
- Післяумови (Postconditions): Умови, які мають бути істинними *після* завершення операції. Це обов'язок самої реалізації. Наприклад, після `push(S, x)` післяумовою є те, що `isEmpty(S)` — `false`.
- Інваріанти класу: Властивості, які мають залишатися істинними для об'єкта протягом усього його життя, до і після виклику будь-якої публічної операції. Для збалансованого дерева інваріантом є, наприклад, «висота лівого та правого піддерев відрізняється не більше, ніж на 1».

Ці ідеї дозволяють не тільки писати більш надійний код, але й формально доводити його коректність. Формальна верифікація — це процес використання математичних методів для доведення того, що реалізація структури даних повністю відповідає аксіомам її АТД. Для цього існують спеціалізовані мови та інструменти (Coq, Dafny, Spec#).

3.6. Приклад повної специфікації: АТД «Множина» (Set)

- Множина значень: Неупорядкована колекція унікальних елементів довільного типу T.
- Операції:
  - `insert(S, x)`: додати елемент `x` до множини `S`.
  - `delete(S, x)`: видалити елемент `x` з множини `S`, якщо він існує.
  - `contains(S, x)`: перевірити, чи належить `x` множині `S`.
  - `size(S)`: повернути кількість елементів (кардинальність) множини.
- Аксіоми:
  1. Для будь-яких `S`, `x`: `contains(insert(S, x), x)` є `true`.

2. Для будь-яких `S`, `x`, `y` (де `x != y`): `contains(insert(S, y), x)` є те ж саме, що `contains(S, x)`.
3. Для будь-яких `S`, `x`: `contains(delete(S, x), x)` є `false`.
4. `size(insert(S, x))` дорівнює `size(S)`, якщо `contains(S, x)` є `true`; інакше дорівнює `size(S) + 1`.

– Можливі реалізації:
  - Hash Set: на основі хеш-таблиці. Середній час операцій — $O(1)$. Порядок елементів не зберігається.
  - Tree Set: на основі збалансованого дерева пошуку (напр., червоно-чорного). Час операцій — $O(\log n)$. Гарантує можливість ітерації по елементах у відсортованому порядку.

3.7. Параметричний поліморфізм і generic-контейнери

АТД за своєю природою є абстрактними не тільки від реалізації, але й від конкретного типу даних, що зберігаються. Сучасні мови програмування підтримують цю ідею через параметричний поліморфізм, більш відомий як generics (узагальнення).

Це дозволяє нам написати код для структури даних один раз, а потім використовувати його для зберігання будь-яких типів даних, зберігаючи при цьому строгий контроль типів на етапі компіляції.

– У C++: `template<class T> class vector;`
– У Java/C#: `class ArrayList<E>`
– У Rust: `struct Vec<T>`

Таким чином, ми можемо створити `vector<int>`, `vector<string>` або `vector<MyCustomObject>`, і компілятор гарантує, що ми не зможемо випадково додати рядок до вектора цілих чисел. Це потужний інструмент, що поєднує гнучкість та безпеку.

1.2. Аналіз складності алгоритмів

Цей розділ закладає основи для розуміння ефективності алгоритмів — навички, яка є центральною для будь-якого професійного розробника. Ми детально розберемо, як можна об’єктивно порівнювати алгоритми та прогнозувати їхню поведінку.

1. Теоретичні відомості

1.1. Навіщо аналізувати алгоритми? Інтуїтивне порівняння

Уявіть, що два шеф-кухарі отримали однакове завдання: приготувати страву за новим рецептом. Перший впорався за 20 хвилин, другий — за 40. Чи означає це, що рецепт першого кухаря вдвічі кращий? Не обов'язково. Можливо, перший кухар — досвідчений професіонал з найсучаснішим обладнанням, а другий — новачок на старій кухні. Щоб об'єктивно порівняти рецепти (алгоритми), нам потрібен спосіб, незалежний від виконавця (процесора), інструментів (апаратного забезпечення) та навіть якості інгредієнтів (вхідних даних).

Просто виміряти час виконання програми — це поганий спосіб оцінки алгоритму, оскільки результат буде залежати від:

- Потужності комп'ютера: На швидшому процесорі будь-яка програма працюватиме швидше.
- Мови програмування та компілятора: Одна й та сама логіка може бути реалізована з різною ефективністю.
- Вхідних даних: Алгоритм сортування може миттєво обробити вже відсортований масив, але довго працювати з масивом, відсортованим у зворотному порядку.

Нам потрібен універсальний математичний інструмент, що дозволяє оцінити внутрішню ефективність алгоритму, його здатність масштабуватися при збільшенні обсягу даних. Цей інструмент — асимптотичний аналіз складності.

1.2. Поняття обчислювальної складності: Час та Простір

Коли ми говоримо про «складність» алгоритму, ми зазвичай маємо на увазі два ресурси, які він споживає:

1. Часова складність (Time Complexity): Це оцінка того, як зростає кількість базових операцій, які виконує алгоритм, залежно від розміру вхідних даних. «Базова операція» — це елементарна дія, як-от арифметична операція, порівняння, присвоєння значення.
2. Просторова складність (Space Complexity): Це оцінка того, як зростає обсяг додаткової пам'яті, яку використовує алгоритм (окрім пам'яті для зберігання вхідних даних), залежно від розміру цих даних.

Хоча обидва показники важливі, часова складність традиційно вважається більш критичною, і саме на ній ми зосередимось.

1.3. **Асимптотичний аналіз: погляд у нескінченність**

Ключова ідея асимптотичного аналізу полягає в тому, щоб зрозуміти, як поводитиметься алгоритм, коли розмір вхідних даних (n) прямує до нескінченності. Нас не так цікавить точна кількість операцій (наприклад, 5n + 12), як характер зростання цієї кількості. Чи зростає вона лінійно, квадратично, логарифмічно? Це дозволяє нам порівнювати алгоритми в їхньому граничному стані.

Для цього використовуються спеціальні математичні позначення, найважливішим з яких є нотація «Велике О» (Big O).

1.4. Велике О (Big O): Верхня межа (оцінка найгіршого випадку)

Нотація Big O описує верхню межу зростання функції. Говорячи простіше, це гарантія: «Мій алгоритм для n елементів ніколи не виконає більше, ніж приблизно $g(n)$ операцій». Вона описує найгірший можливий сценарій поведінки алгоритму.

Формально, $f(n) = O(g(n))$ означає, що існують такі додатні константи C та n_0, що для всіх $n \geq n_0$ виконується нерівність $0 \leq f(n) \leq C \cdot g(n)$.

Давайте розглянемо основні класи складності на прикладах.

- $O(1)$ — Константний час
    - Ідея: Час виконання алгоритму не залежить від розміру вхідних даних.
    - Аналогія: Взяти першу книгу зі стопки. Неважливо, скільки книг у стопці (10 чи 1000), ця операція займає однаковий час.
    - Приклад: Доступ до елемента масиву за індексом.

```
ФУНКЦІЯ GetElement(Array A, index):
    ПОВЕРНУТИ A[index]
КІНЕЦЬ ФУНКЦІЇ
```

- $O(\log n)$ — Логарифмічний час
    - Ідея: На кожному кроці алгоритм відкидає значну частину (зазвичай половину) даних, що залишились.
    - Аналогія: Пошук слова у паперовому словнику. Ви відкриваєте його посередині, відкидаєте половину, потім знову відкриваєте залишок посередині, тощо.
    - Приклад: Бінарний пошук у відсортованому масиві.

```
ФУНКЦІЯ BinarySearch(Array A, key):
```

```
    left = 0, right = довжина(A) - 1
    ПОКИ left <= right:
        mid = left + (right - left) / 2
        ЯКЩО A[mid] == key ТО ПОВЕРНУТИ mid
        ЯКЩО A[mid] < key ТО left = mid + 1
        ІНАКШЕ right = mid - 1
    КІНЕЦЬ ПОКИ
    ПОВЕРНУТИ -1 // Не знайдено
КІНЕЦЬ ФУНКЦІЇ
```

– $O(n)$ — Лінійний час

- Ідея: Алгоритм виконує одну і ту ж роботу для кожного елемента вхідних даних.
- Аналогія: Прочитати кожну сторінку в книзі. Чим більше сторінок, тим більше часу це займе, і залежність пряма.
- Приклад: Пошук максимального елемента в масиві.

```
ФУНКЦІЯ FindMax(Array A):
    max_element = A[0]
    ДЛЯ i ВІД 1 ДО довжина(A)-1:
        ЯКЩО A[i] > max_element ТО max_element = A[i]
    КІНЕЦЬ ДЛЯ
    ПОВЕРНУТИ max_element
КІНЕЦЬ ФУНКЦІЇ
```

– $O(n\log n)$ — Лінійно-логарифмічний час

- Ідея: Це «золотий стандарт» для ефективних алгоритмів сортування. Алгоритм розбиває задачу на частини, а потім об’єднує їх.
- Аналогія: Ефективно відсортувати велику колоду карт, постійно ділячи її на дві частини, сортуючи кожну, а потім зливаючи відсортовані половинки разом.
- Приклад: Сортування злиттям (Merge Sort), Швидке сортування (QuickSort) в середньому випадку.

– $O(n^2)$ — Квадратичний час

- Ідея: Алгоритм порівнює кожен елемент з кожним іншим елементом. Зазвичай це реалізується через вкладені цикли.
- Аналогія: У кімнаті з n людьми кожен тисне руку кожному іншому. Кількість рукостискань зростає дуже швидко.

- Приклад: Прості алгоритми сортування, як-от сортування бульбашкою.

```
ФУНКЦІЯ BubbleSort(Array A):
    n = довжина(A)
    ДЛЯ i ВІД 0 ДО n-2:
        ДЛЯ j ВІД 0 ДО n-i-2:
            ЯКЩО A[j] > A[j+1] ТО
                Поміняти_місцями(A[j], A[j+1])
            КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ФУНКЦІЇ
```

– $O(2^n)$ — Експоненційний час

- Ідея: Для кожного елемента існує два варіанти вибору, і алгоритм перебирає всі можливі комбінації.
- Аналогія: Підбір n-значного бінарного пароля шляхом перебору всіх можливих 2^n комбінацій.
- Приклад: Наївне рекурсивне обчислення чисел Фібоначчі.

– $O(n!)$ — Факторіальний час

- Ідея: Алгоритм перебирає всі можливі перестановки елементів.
- Аналогія: Розв'язання задачі комівояжера шляхом перевірки кожного можливого маршруту.
- Приклад: Генерація всіх перестановок рядка.

1.5. Інші нотації: Ω (Омега) та Θ (Тета)

Для повного розуміння варто знати ще дві нотації:

– $\Omega(g(n))$ — Нижня межа (оцінка найкращого випадку): Це гарантія, що алгоритм буде працювати не швидше, ніж *g(n)*. Наприклад, навіть якщо нам пощастить і ми шукаємо перший елемент у масиві, лінійний пошук все одно виконає одну операцію, тому його складність $\Omega(1)$.

– $\Theta(g(n))$ — Точна асимптотична оцінка: Це найсильніше твердження. Воно означає, що алгоритм «затиснутий» між верхньою та нижньою межами, і його ріст точно відповідає *g(n)*. Наприклад, пошук максимального елемента в масиві завжди вимагає перегляду всіх n елементів, тому його складність $\Theta(n)$.

1.6. Аналіз на практиці: найкращий, середній та найгірший випадки

Аналізуючи алгоритм, важливо розрізняти три сценарії:

- Найкращий випадок: Вхідні дані, для яких алгоритм працює найшвидше.
- Найгірший випадок: Вхідні дані, для яких алгоритм працює найповільніше. Саме цей випадок зазвичай описує нотація Big O.
- Середній випадок: Усереднена поведінка на всіх можливих вхідних даних.

Приклад: Швидке сортування

- *Найкращий та середній випадки:* $O(n\log n)$. Це відбувається, коли опорний елемент щоразу ділить масив приблизно навпіл.
- *Найгірший випадок:* $O(n^2)$. Це відбувається, коли масив вже відсортований, і опорний елемент щоразу виявляється найбільшим або найменшим.

1.7. Практичні методи аналізу складності

Тепер, коли ми знайомі з основними поняттями асимптотичної складності, розглянемо, як застосовувати цей аналіз до двох основних типів алгоритмів: ітеративних та рекурсивних.

1.7.1. Аналіз ітеративних алгоритмів

Аналіз ітеративних алгоритмів, тобто алгоритмів, що використовують цикли (**`for`**, **`while`**), зводиться до підрахунку загальної кількості ітерацій, які виконують ці цикли.

- Прості послідовні цикли

  Якщо алгоритм складається з кількох циклів, що виконуються послідовно (один за одним), їхня загальна складність дорівнює складності «найважчого» (найдовшого) з них.

```
// Цикл 1: O(n)
ДЛЯ і ВІД 1 ДО n:
    // якась O(1) операція
КІНЕЦЬ ДЛЯ

// Цикл 2: O(m)
ДЛЯ j ВІД 1 ДО m:
    // якась O(1) операція
КІНЕЦЬ ДЛЯ
```

Загальна складність такого коду буде $O(n + m)$. Якщо n m, то $O(n)$.

– Вкладені цикли

Якщо один цикл вкладений в інший, їхні складності перемножуються.

```
// Зовнішній цикл виконується n разів
ДЛЯ i ВІД 1 ДО n:
    // Внутрішній цикл теж виконується n разів для КОЖНОЇ ітерац
ії зовнішнього
    ДЛЯ j ВІД 1 ДО n:
        // якась O(1) операція
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ДЛЯ
```

Загальна кількість операцій тут буде $n \times n = n^2$. Отже, складність — $O(n^2)$.

– Вкладені цикли із залежними ітераторами

Часто зустрічається випадок, коли кількість ітерацій внутрішнього циклу залежить від ітератора зовнішнього.

```
ДЛЯ i ВІД 1 ДО n:
    ДЛЯ j ВІД i ДО n:
        // якась O(1) операція
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ДЛЯ
```

Давайте порахуємо кількість ітерацій внутрішнього циклу:

- Коли `i = 1`, `j` пробігає від 1 до *n* (*n* ітерацій).
- Коли `i = 2`, `j` пробігає від 2 до *n* (*n-1* ітерацій).
- …
- Коли `i = n`, `j` пробігає від *n* до *n* (1 ітерація).

Загальна кількість операцій — це сума арифметичної прогресії: $n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$. При асимптотичному аналізі ми відкидаємо константи та члени нижчого порядку, отже складність такого алгоритму все одно є $O(n^2)$.

– Логарифмічні цикли

Якщо ітератор циклу не збільшується на константу, а множиться (або ділиться) на неї, складність стає логарифмічною.

```
i = 1
ПОКИ i < n:
    // якась O(1) операція
```

```
    i = i * 2
КІНЕЦЬ ПОКИ
```

Змінна **i** приймає значення 1, 2, 4, 8, 16, …, 2^k. Цикл завершиться, коли $2^k \geq n$, або $k \geq \log_2 n$. Таким чином, кількість ітерацій дорівнює приблизно $\log_2 n$, і складність такого циклу — $O(\log n)$.

1.7.2. Аналіз рекурсивних алгоритмів

Аналіз рекурсивних алгоритмів є складнішим, оскільки потрібно оцінити загальну кількість операцій, що виконуються у всіх рекурсивних викликах. Ключовим інструментом для цього є рекурентне співвідношення.

Рекурентне співвідношення — це рівняння, яке виражає складність задачі розміром n, $T(n)$, через її ж складність для менших вхідних даних.

Наприклад, для бінарного пошуку: $T(n) = T(n/2) + O(1)$

- $T(n/2)$: вартість одного рекурсивного виклику для підзадачі, що вдвічі менша (ми відкидаємо половину масиву).
- $O(1)$: вартість роботи на поточному кроці (одне порівняння).

Існує декілька методів для розв'язання таких співвідношень:

- Метод підстановки: Ми «вгадуємо» відповідь і доводимо її коректність за допомогою методу математичної індукції.
- Дерево рекурсії: Ми візуалізуємо рекурсивні виклики у вигляді дерева, обчислюємо вартість роботи на кожному рівні, а потім підсумовуємо вартість по всіх рівнях. Це дуже наочний метод.
- Майстер-теорема: Потужний інструмент, який дає готовий розв'язок для великого класу рекурентних співвідношень.

1.8. Майстер-теорема: «кулінарна книга» для рекурсій

Майстер-теорема (часто її називають «основною теоремою про рекурсій») є потужним «шорткатом» для аналізу алгоритмів, що працюють за принципом «Розділяй і володарюй». Вона дозволяє отримати розв'язок, просто подивившись на структуру співвідношення.

Теорема застосовується до рекурентних співвідношень вигляду:

$$T(n) = aT(n/b) + f(n)$$

Пояснення позначень:

- $T(n)$: Час роботи для задачі розміром n.

– a: Кількість підзадач, на які розбивається початкова задача (кількість рекурсивних викликів). $a \geq 1$.
– n/b: Розмір кожної підзадачі. b 1.
– $f(n)$: Вартість роботи, виконаної поза рекурсивними викликами (тобто вартість кроків «Розділяй» та «Комбінуй»).

Суть теореми полягає у порівнянні швидкості зростання функції $f(n)$ з функцією $n^{\log_b a}$. Ця величина представляє вартість роботи, що виконується у листках дерева рекурсії. Залежно від того, яка з цих двох функцій «важча», ми отримуємо три можливі випадки.

Випадок 1: Робота в листках домінує Якщо функція $n^{\log_b a}$ зростає суттєво швидше, ніж $f(n)$.

– Формально: Якщо $f(n) = O(n^{\log_b a - \epsilon})$ для деякої константи ϵ 0.
– Результат: $T(n) = \Theta(n^{\log_b a})$
– Приклад: Алгоритм Штрассена: $T(n) = 7T(n/2) + O(n^2)$.
  - Тут $a = 7, b = 2, f(n) = n^2$.
  - Обчислюємо $n^{\log_2 7} \approx n^{2.807}$.
  - Бачимо, що $n^{2.807}$ зростає швидше за $f(n) = n^2$. Випадок 1 застосовується.
  - Складність: $\Theta(n^{\log_2 7})$.

Випадок 2: Робота розподілена рівномірно Якщо швидкість зростання $f(n)$ така ж, як і у $n^{\log_b a}$.

– Формально: Якщо $f(n) = \Theta(n^{\log_b a})$.
– Результат: $T(n) = \Theta(n^{\log_b a} \log n)$
– Приклад: Сортування злиттям: $T(n) = 2T(n/2) + O(n)$.
  - Тут $a = 2, b = 2, f(n) = n$.
  - Обчислюємо $n^{\log_2 2} = n^1 = n$.
  - Бачимо, що $f(n) = n$ має такий самий порядок росту, як і n. Випадок 2 застосовується.
  - Складність: $\Theta(n \log n)$.

Випадок 3: Робота в корені домінує Якщо функція $f(n)$ зростає суттєво швидше, ніж $n^{\log_b a}$.

- Формально: Якщо $f(n) = \Omega(n^{\log_b a+\epsilon})$ для деякої константи $\epsilon\, 0$, і додатково виконується «умова регулярності» $a \cdot f(n/b) \leq c \cdot f(n)$ для деякої константи $c < 1$.
- Результат: $T(n) = \Theta(f(n))$
- Приклад: Гіпотетичний алгоритм: $T(n) = 3T(n/4) + O(n^2)$.
  - Тут $a = 3, b = 4, f(n) = n^2$.
  - Обчислюємо $n^{\log_4 3} \approx n^{0.79}$.
  - Бачимо, що $f(n) = n^2$ зростає значно швидше за $n^{0.79}$. Умова регулярності також виконується. Випадок 3 застосовується.
  - Складність: $\Theta(n^2)$.

Майстер-теорема є надзвичайно корисним інструментом, але варто пам'ятати, що вона покриває не всі можливі рекурентні співвідношення. Існують «проміжки» між її випадками.

1.9. Амортизаційна складній. Реальний підхід до аналізу.

Амортизаційна складність — це метод аналізу, який оцінює середню вартість операції в послідовності операцій, а не вартість однієї операції в найгіршому випадку.

Уявіть, що ви користуєтесь проїзним на громадський транспорт, який коштує 600 гривень на місяць і дає право на необмежену кількість поїздок.

Аналіз найгіршого випадку: Ваша перша поїздка в місяці "коштує" вам 600 гривень. Це дуже дорого!

Амортизаційний аналіз: Якщо ви робите 60 поїздок на місяць, ви можете "розмазати" вартість проїзного на всі поїздки. Амортизована вартість кожної поїздки становить $600/60 = 10$ гривень.

Амортизаційна складність працює за тим самим принципом. Вона розглядає послідовність операцій, де більшість операцій є дуже "дешевими", але зрідка трапляється одна дуже "дорога". Замість того, щоб казати, що складність дорівнює вартості найдорожчої операції, ми "розмазуємо" її вартість на всі дешеві операції, що їй передували.

Це дає нам більш реалістичну та справедливу оцінку середньої продуктивності. Традиційний аналіз найгіршого випадку (Big O) іноді може бути занадто "песимістичним". Він фокусується на одній, потенційно дуже рідкісній, але дорогій операції, і поширює її вартість на всі інші.

Розглянемо динамічний масив (вектор). Якщо ми скажемо, що складність додавання елемента в кінець — це $O(n)$, це буде формально правильно (адже іноді нам доводиться копіювати весь масив), але водночас це вводить в оману. На практиці 99% операцій додавання виконуються миттєво, за $O(1)$.

Амортизаційний аналіз дозволяє математично обґрунтувати, чому, незважаючи на рідкісні дорогі операції, середня продуктивність залишається надзвичайно високою. Він потрібен, щоб:

1. Отримати більш точну характеристику продуктивності для структур даних, що мають рідкісні "пікові" навантаження.

2. Обґрунтувати вибір таких структур, як динамічний масив або хеш-таблиця, які є надзвичайно швидкими на практиці.

3. Відрізняти гарантовану продуктивність кожної операції від середньої продуктивності в довгій послідовності.

Розглянемо ще раз приклад з масивом: динамічний масив.

Давайте детально проаналізуємо операцію додавання елемента в кінець (`append`) динамічного масиву, який при заповненні подвоює свою місткість.

- "Дешева" операція: Якщо в масиві є вільне місце (`size < capacity`), ми просто додаємо елемент. Вартість — 1 умовна одиниця.
- "Дорога" операція: Якщо масив заповнений (`size == capacity`), нам потрібно:
    1. Виділити новий масив вдвічі більшого розміру (нехай `new_capacity = 2 * capacity`).
    2. Скопіювати всі `n` існуючих елементів у новий масив. Вартість — n одиниць.
    3. Вставити новий елемент. Вартість — 1 одиниця. Загальна вартість цієї операції — n + 1 одиниць.

Уявімо, що ми починаємо з порожнього масиву з місткістю 1.

Операція	Дія	Вартість	Поточний розмір	Поточна місткість
1	**append**	1	1	1
2	**append** (перерозподіл)	1 (копіювання) + 1 (вставка) = 2	2	2
3	**append** (перерозподіл)	2 (копіювання) + 1 (вставка) = 3	3	4
4	**append**	1	4	4
5	**append** (перерозподіл)	4 (копіювання) + 1 (вставка) = 5	5	8
6	**append**	1	6	8
7	**append**	1	7	8
8	**append**	1	8	8
9	**append** (перерозподіл)	8 (копіювання) + 1 (вставка) = 9	9	16

Подивимось на загальну вартість 9 операцій:

1 + 2 + 3 + 1 + 5 + 1 + 1 + 1 + 9 = 24.

Середня (амортизована) вартість однієї операції: **24 / 9 = 2.66**.

Як бачимо, хоч у нас і були дуже дорогі операції вартістю 5 і 9, середня вартість виявилася дуже низькою, близькою до константи. Можна математично довести, що:

для *n* операцій додавання загальна вартість ніколи не перевищить *3n*.

Отже, амортизована вартість однієї операції — $O(3n/n) = O(3) = O(1)$.

Ми “заплатили” за дорогі операції заздалегідь, виконуючи дешеві.

Розуміння амортизаційної складності дає програмісту можливість приймати більш обґрунтовані рішення при виборі структур даних.

1. Впевненість у середній продуктивності: Коли ви бачите, що амортизаційна складність операції **append** для вектора або **insert** для хеш-таблиці дорівнює $O(1)$, ви можете бути впевнені, що навіть при виконанні мільйона таких операцій середня швидкість буде дуже високою.
2. Розуміння компромісів: Ви розумієте, що хоча середня швидкість висока, іноді операція може бути повільною. Це важливо для систем реального часу, де навіть рідкісні затримки є неприпустимими. У такому випадку, можливо, доведеться обрати іншу структуру (наприклад, зв'язний список), яка має гіршу середню, але стабільну і передбачувану продуктивність у найгіршому випадку.
3. Правильна оцінка коду: Ви не будете відмовлятися від використання динамічного масиву лише тому, що в його документації написано "складність вставки у найгіршому випадку — $O(n)$". Ви будете знати, що цей найгірший випадок компенсується великою кількістю надзвичайно швидких операцій.

Таким чином, амортизаційна складність — це інструмент, що дає більш реалістичну і практичну оцінку продуктивності алгоритмів, поведінка яких не є рівномірною.

2. Область застосування

Аналіз складності є універсальним інструментом в арсеналі інженера та науковця.

- Проєктування ПЗ: Для вибору найбільш відповідного алгоритму та структури даних на етапі проєктування системи.
- Наукові дослідження: Для доведення ефективності нових розроблених алгоритмів.
- Прогнозування продуктивності: Для оцінки того, як система буде поводитися під високим навантаженням.
- Співбесіди: Це стандартна тема на технічних співбесідах у провідних IT-компаніях.

3. Приклади застосування

- Бази даних: При проєктуванні індексів обирають B+-дерева, оскільки їхні логарифмічні операції $O(\log n)$ гарантують швидкий пошук навіть у мільярдах записів.

– Пошукові системи: Використовують структури даних, близькі до хеш-таблиць (інвертовані індекси), щоб забезпечити пошук за час, близький до $O(1)$.
– Комп'ютерна графіка: Для рендерингу складних 3D-сцен у реальному часі алгоритми мають працювати за час, близький до лінійного, інакше гравець помітить падіння FPS.

Переваги та недоліки асимптотичного аналізу

Переваги:

– Об'єктивність: Не залежить від заліза, мови програмування чи компілятора.
– Простота порівняння: Дозволяє легко порівнювати принципово різні підходи.
– Фокус на масштабованості: Показує, як алгоритм буде поводитися на великих даних, що є найважливішим у більшості реальних систем.

Недоліки:

– Ігнорування констант: Алгоритм зі складністю $O(n)$ може бути на практиці повільнішим за алгоритм $O(n^2)$ для малих n, якщо у першого дуже великий прихований коефіцієнт.
– Не враховує апаратні особливості: Аналіз не враховує вплив кеш-пам'яті, конвеєризації процесора та інших апаратних оптимізацій.
– Найгірший випадок може бути рідкісним: Іноді алгоритм, що має погану складність у найгіршому випадку, на практиці працює дуже швидко, оскільки цей найгірший випадок майже ніколи не трапляється.

Контрольні запитання

1. Поясніть своїми словами, чому вимірювання часу виконання програми не є надійним способом оцінки ефективності алгоритму.
2. Що таке часова та просторова складність? Яка з них зазвичай є більш важливою?
3. Що описує нотація Big O? Наведіть аналогію для $O(\log n)$ та $O(n^2)$.
4. Розташуйте наступні класи складності у порядку зростання: $O(n)$, $O(n^2)$, $O(1)$, $O(\log n)$, $O(n \log n)$, $O(n)$, $O(2^n)$.
5. Наведіть приклад алгоритму, у якого складність у найкращому та найгіршому випадках кардинально відрізняється.

Методичні вказівки до лабораторної роботи

1. Назва

Емпіричне дослідження та порівняння часової складності алгоритмів пошуку.

2. Мета

Практично переконатись у теоретичних оцінках складності. Навчитись вимірювати час виконання програм та візуалізувати результати, щоб побачити різницю між лінійною ($O(n)$) та логарифмічною ($O(\log n)$) складністю.

3. Завдання

Реалізувати два алгоритми пошуку елемента в масиві: лінійний пошук та бінарний пошук. Провести серію експериментів для вимірювання часу їх роботи на масивах різного розміру та візуалізувати отримані дані у вигляді графіка.

4. Опис

Програма повинна генерувати масиви різного розміру, заповнювати їх даними (для бінарного пошуку масив має бути відсортований), а потім багато разів запускати пошук елемента, заміряючи середній час виконання.

5. Порядок виконання

1. Реалізуйте алгоритми: Напишіть функції для лінійного та бінарного пошуку.
2. Створіть генератор даних: Напишіть функцію, що створює та заповнює масив заданого розміру `N`.
3. Напишіть експериментальну функцію:
    - Вона приймає розмір масиву `N`.
    - Створює масив та сортує його.
    - Запускає, наприклад, 1000 разів лінійний пошук (шукаючи випадковий елемент) і заміряє загальний час.
    - Аналогічно запускає 1000 разів бінарний пошук і заміряє час.
    - Повертає середній час для кожної з операцій.
4. Проведіть серію експериментів: Викличте вашу експериментальну функцію для різних значень `N` (напр., 100, 1000, 10000, 100000, 1000000).

5. Візуалізуйте результати: Запишіть отримані дані в таблицю. Використовуючи будь-який інструмент (Excel, Python matplotlib), побудуйте графік, де по осі X буде розмір масиву `N`, а по осі Y — час виконання. Ви маєте побачити дві криві, що кардинально відрізняються.

6. Результат

Таблиця та графік, що наочно демонструють лінійне зростання часу для лінійного пошуку та логарифмічне — для бінарного.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, що реалізує обидва алгоритми та проводить вимірювання.
2. Таблицю та графік з результатами експериментів.
3. Бути готовим відповісти на питання:
    - Чому отримана крива для лінійного пошуку є прямою, а для бінарного — майже горизонтальною?
    - Чому для бінарного пошуку масив обов'язково має бути відсортованим?
    - Як би змінився графік, якби ми вимірювали просторову складність цих алгоритмів?

Тести

1. Який клас складності є найбільш ефективним (найшвидшим) для великих n?

a) $O(n)$

b) $O(n^2)$

c) $O(\log n)$

d) $O(n \log n)$

2. Big O нотація описує:

e) Точний час виконання програми в секундах.

f) Верхню межу зростання часу виконання у найгіршому випадку.

g) Кількість рядків коду в програмі.

h) Нижню межу зростання часу виконання у найкращому випадку.

3. Складність алгоритму, що складається з двох послідовних циклів по n ітерацій кожен, буде:

i) $O(n^2)$

j) $O(2n)$

k) $O(n)$

l) $O(\log n)$

4. Який з наведених алгоритмів має складність $O(1)$?

m) Пошук елемента у відсортованому масиві.

n) Видалення елемента з середини зв'язного списку.

o) Додавання елемента на вершину стека.

p) Сортування масиву.

5. Алгоритм має складність $T(n) = 3T(n/2) + O(n)$. Яка його асимптотична оцінка за майстер-теоремою?

q) $O(n \log n)$

r) $O(n^2)$

s) $O(n^{\log_2 3})$

t) $O(n)$

Відповіді: 1-c, 2-b, 3-c (бо $O(n + n) = O(2n) = O(n)$), 4-c, 5-c

1. Якою є часова складність (Big O) для наступного фрагмента коду?

```
count = 0
ДЛЯ i ВІД 1 ДО n:
    ДЛЯ j ВІД 1 ДО i:
        count = count + 1
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ДЛЯ
```

u) $O(n)$

v) $O(n \log n)$

w) $O(n^2)$

x) $O(n^3)$

2. Яка часова складність (Big O) для наступного фрагмента коду?

```
i = n
count = 0
ПОКИ i > 1:
    i = i / 2
```

```
    count = count + 1
КІНЕЦЬ ПОКИ
```

y) $O(n)$

z) $O(\log n)$

aa) $O(n^2)$

bb) $O(\sqrt{n})$

3. Використовуючи майстер-теорему, визначте асимптотичну складність для рекурентного співвідношення $T(n) = 8T(n/2) + O(n^2)$.

cc) $\Theta(n^2)$

dd) $\Theta(n^2 \log n)$

ee) $\Theta(n^3)$

ff) $\Theta(n^{\log_2 8} \log n)$

4. Який випадок майстер-теореми застосовується до рекурентного співвідношення для сортування злиттям: $T(n) = 2T(n/2) + O(n)$?

gg) Випадок 1 (робота в листках домінує)

hh) Випадок 2 (робота розподілена рівномірно)

ii) Випадок 3 (робота в корені домінує)

jj) Майстер-теорема не може бути застосована до цього співвідношення.

5. Рекурентне співвідношення $T(n) = 2T(n/2) + O(n^2)$ описує алгоритм, де:

kk) Вартість рекурсивних викликів є домінуючою.

ll) Вартість роботи на кожному рівні дерева рекурсії однакова.

mm) Вартість кроків «розділяй» та «комбінуй» на кожному рівні є домінуючою.

nn) Алгоритм має логарифмічну складність.

Відповіді: 1-c, 2-b, 3-c (бо $n^{\log_2 8} = n^3$, що є «важчим» за n^2, Випадок 1), 4-b (бо $n^{\log_2 2} = n^1$, що є того ж порядку, що і $f(n) = n$), 5-c

Завдання для самостійної роботи

1. Аналіз коду. Визначте асимптотичну складність (Big O) для кожного з наведених фрагментів псевдокоду.

2. Просторова складність. Оцініть просторову складність для алгоритмів лінійного та бінарного пошуку.
3. Найкращий/найгірший випадок. Опишіть найкращий та найгірший сценарії для алгоритму пошуку вставки. Якою буде складність у цих випадках?
4. Ранжування функцій. Розташуйте наступні функції у порядку зростання їх асимптотичної складності: n, $1000n^2$, $n\log n$, 2^n, n^3, $\log n$, $n^{2.5}$.
5. Два масиви. Якою буде складність алгоритму, що перевіряє, чи є в двох масивах (розміром M та N) спільні елементи, за допомогою вкладених циклів?
6. Оптимізація. Як можна покращити алгоритм з попереднього завдання, використовуючи хеш-таблицю? Якою стане його складність?
7. Аналіз рекурсії. Визначте складність рекурсивної функції, що обчислює факторіал.
8. Матриці. Якою є складність додавання двох матриць розміром $n \times n$?
9. Дерева. Якою є часова складність повного обходу (відвідування кожної вершини) збалансованого бінарного дерева з n вузлами?
10. Пошук пари. Вам дано масив чисел та цільове значення `sum`. Напишіть алгоритм зі складністю $O(n\log n)$, що визначає, чи існує в масиві пара елементів, сума яких дорівнює `sum`.
11. Що «ховає» Big O? Наведіть гіпотетичний приклад двох алгоритмів A та B, де A має складність $O(n)$, а B — $O(n^2)$, але для N=100 алгоритм B працює швидше. Поясніть, чому так може бути.
12. Складність вашого улюбленого алгоритму. Оберіть будь-який відомий вам алгоритм (окрім розглянутих) і детально проаналізуйте його часову та просторову складність.
13. Амортизаційний аналіз. Дослідіть, що таке амортизаційний аналіз, і поясніть на прикладі додавання елемента в динамічний масив (вектор), чому ця операція має амортизаційну складність $O(1)$.
14. Майстер-теорема. Знайдіть та опишіть всі три випадки майстер-теореми для розв’язання рекурентних співвідношень.

15. Есе-роздум. Напишіть есе на тему: «Чи завжди варто обирати алгоритм з найкращою асимптотичною складністю в реальних проєктах?».

16. Аналіз складності ітеративного алгоритму. Проведіть детальний аналіз часової складності (Big O) для наступного фрагмента псевдокоду. Поясніть, як ви отримали результат, враховуючи вкладеність та залежність циклів.

```
ФУНКЦІЯ AnalyzeComplexity(Array A):
    n = довжина(A)
    sum = 0
    // Блок 1
    ДЛЯ i ВІД 1 ДО n:
        sum = sum + A[i-1]
    КІНЕЦЬ ДЛЯ

    // Блок 2
    ДЛЯ i ВІД 1 ДО n кроком i=i*3:
        // Блок 3
        ДЛЯ j ВІД 1 ДО i:
            sum = sum - 1
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ДЛЯ
    ПОВЕРНУТИ sum
КІНЕЦЬ ФУНКЦІЇ
```

17. Розв’язання рекурентності методом дерева рекурсії. Розгляньте рекурентне співвідношення $T(n) = 2T(n/3) + O(n^2)$. Побудуйте дерево рекурсії для цього співвідношення, визначте вартість роботи на кожному рівні та загальну кількість рівнів. Просумуйте вартість по всіх рівнях, щоб отримати фінальну асимптотичну оцінку для $T(n)$.

18. Практичне застосування майстер-теореми. Для кожного з наведених нижче рекурентних співвідношень визначте, чи може бути застосована майстер-теорема. Якщо так, вкажіть, який випадок теореми застосовується, та надайте розв’язок у нотації Θ.

- a) $T(n) = 4T(n/2) + n^2$
- b) $T(n) = 4T(n/2) + n^3$
-

c) $T(n) = 16T(n/4) + n^2 \log n$

19. Аналіз просторової складності. Порівняйте просторову складність (додаткову пам'ять, що використовується) для двох реалізацій обчислення факторіалу: ітеративної та рекурсивної. Поясніть, чому рекурсивна версія зазвичай вимагає більше пам'яті.

20. Задача про пошук пари. Вам дано масив унікальних цілих чисел та цільове значення `sum`. Запропонуйте два алгоритми, які визначають, чи існує в масиві пара елементів, сума яких дорівнює `sum`.
    - Алгоритм 1: Наївний, що перебирає всі можливі пари.
    - Алгоритм 2: Більш ефективний, що використовує сортування або хеш-таблицю. Проаналізуйте та порівняйте часову складність обох запропонованих вами алгоритмів.

Базові лінійні структури диних

Порядок та Послідовність як основа організації даних

У попередніх розділах ми встановили, що алгоритми — це рецепти, а структури даних — це способи організації інгредієнтів. Тепер настав час перейти від загальних концепцій до конкретних інструментів. Ми почнемо наш шлях з найпростіших та найінтуїтивніших способів організації даних — лінійних структур.

Що спадає на думку, коли ми чуємо слово "послідовність"? Список покупок, черга людей, низка намистин, вагони потяга. Всі ці об'єкти об'єднує одна фундаментальна властивість: їхні елементи розташовані один за одним у чіткому, лінійному порядку. У кожного елемента (окрім першого та останнього) є рівно один попередник і рівно один наступник. Це і є суть лінійності.

У цифровому світі ця проста ідея є надзвичайно потужною. Більшість інформації, з якою ми працюємо, має природну послідовну структуру: текст складається з послідовності символів, музичний твір — з послідовності нот, історія транзакцій — з послідовності операцій. Лінійні структури даних надають нам інструменти для ефективного представлення та маніпулювання такими послідовностями.

Однак, як ми скоро побачимо, навіть така проста ідея, як "зберігати елементи по порядку", може бути реалізована кардинально різними способами. Уявіть, що вам потрібно розкласти колекцію книг.

Спосіб 1: Ви можете поставити їх щільно одну до одної на довгій полиці. Це зручно, якщо ви знаєте порядковий номер книги, але якщо вам потрібно вставити нову книгу в середину, доведеться пересунути половину бібліотеки.

Спосіб 2: Ви можете покласти кожну книгу в окрему коробку, а на кожній коробці написати, де знаходиться наступна книга. Тепер вставити нову книгу легко — достатньо лише змінити два "посилання". Але щоб знайти соту книгу, вам доведеться послідовно відкрити 99 попередніх коробок.

Ці два підходи — зберігання в суцільному блоці та зберігання за допомогою зв'язків — є двома фундаментальними філософіями, що лежать в основі лінійних структур даних. У цьому розділі ми детально дослідимо, як ці філософії втілюються у найважливіших структурах: масивах та зв'язаних списках. А потім побачимо, як на їх основі, немов надбудови, створюються більш спеціалізовані структури — стеки та черги, які диктують власні правила доступу до даних.

Опанування цих чотирьох базових структур є абсолютно необхідним першим кроком. Це ті "цеглинки", з яких згодом будуть складатися значно складніші алгоритми та системи.

Дві філософії зберігання: суцільність проти зв'язності

Як комп'ютер може представити послідовність у своїй пам'яті, яка, по суті, є лише гігантським масивом пронумерованих комірок? Існують два фундаментально різні підходи.

Філософія 1: Суцільне зберігання — Масив

Найпростіший спосіб — виділити один великий, неперервний блок пам'яті та розмістити в ньому всі елементи нашої послідовності один за одним. Це все одно, що забронювати ряд крісел у кінотеатрі. Цей підхід лежить в основі масиву (Array).

Як це працює? Коли ви створюєте масив на 100 цілих чисел, операційна система знаходить вільний блок пам'яті, достатній для зберігання цих 100 чисел, і резервує його для вас. Головна перевага такого підходу — миттєвий доступ до будь-якого елемента за його індексом. Оскільки всі елементи мають однаковий розмір і лежать в пам'яті впритул, знайти адресу *i*-го елемента дуже просто:

Адреса(A[i]) = (Початкова адреса масиву A) + i × (розмір одного елемента)

Ця проста арифметична операція виконується процесором миттєво, що і забезпечує складність доступу $O(1)$.

Вбудований компроміс: Проте за цю швидкість доступу доводиться платити. Суцільність пам'яті, яка є головною перевагою масиву, стає його ж головним недоліком, коли послідовність потрібно змінювати.

- Вставка та видалення: Якщо вам потрібно вставити елемент в середину масиву, ви не можете просто "розсунути" пам'ять. Вам доведеться вручну скопіювати всі елементи, що стоять правіше, на одну позицію, щоб звільнити місце. Це повільна операція зі складністю $O(n)$.
- Зміна розміру: Що станеться, якщо у вашому динамічному масиві на 100 елементів закінчиться місце, а вам потрібно додати 101-й? Єдиний вихід — знайти новий, більший блок пам'яті (наприклад, на 200 елементів), скопіювати туди всі 100 старих елементів, а вже потім додати новий. Це теж дорога операція.

Філософія 2: Зв'язне зберігання — Зв'язаний список

А що, якщо відмовитись від ідеї зберігати все разом? Що, якщо кожен елемент нашої послідовності буде окремим, незалежним об'єктом, який просто "знає", де знаходиться наступний? Це філософія зв'язаного списку (Linked List).

Як це працює? Зв'язаний список складається з вузлів (nodes). Кожен вузол — це невелика структура, що містить дві речі: самі дані (наш "інгредієнт") та вказівник (посилання) на наступний вузол у ланцюжку. Ці вузли можуть бути розкидані по всій оперативній пам'яті. Головне — що кожен з них зберігає адресу наступного, утворюючи неперервний логічний ланцюг.

Розв'язання проблеми масиву: Ця архітектура елегантно вирішує проблеми масиву зі вставкою та видаленням. Щоб вставити новий елемент між вузлами `A` та `B`, нам потрібно виконати лише кілька операцій з вказівниками:

1. Створити новий вузол `C`.

2. Зробити так, щоб `C` вказував на `B`.

3. Змінити вказівник `A` так, щоб він вказував на `C`.

Тобто жодного масового копіювання даних. Складність такої вставки — $O(1)$.

Новий компроміс: Але, як ми знаємо, безкоштовних переваг не буває. Розв'язавши проблему вставки, ми створили іншу. Як тепер отримати доступ до 100-го елемента? Єдиний шлях — почати з голови списку і послідовно пройти 99 вказівників, щоб дістатися до потрібного вузла. Це повільна операція зі складністю $O(n)$. Ми втратили миттєвий довільний доступ.

Від зберігання до поведінки: Стеки та Черги як Абстракції

Масиви та зв'язані списки — це фундаментальні способи *фізичного* зберігання даних. Вони надають нам повну свободу дій. Але іноді така свобода не потрібна. Часто в реальному світі процеси підпорядковуються строгим правилам доступу. Для моделювання таких процесів були створені стек (Stack) та черга (Queue).

Важливо зрозуміти, що стек і черга — це не стільки нові структури даних, скільки нові правила гри. Це абстрактні типи даних (АТД), які визначають, *яким чином* дозволено взаємодіяти з послідовністю, обмежуючи повну свободу масивів та списків.

Стек: Принцип LIFO (Last-In, First-Out)

Уявіть стопку свіжовимитих тарілок. Ви кладете чисту тарілку наверх стопки, і коли вам потрібна тарілка, ви теж берете її зверху. Ту, яку поклали останньою. Це і є принцип LIFO — "останнім прийшов, першим пішов".

АТД "Стек" надає лише декілька операцій:

`push(item)`: покласти елемент на вершину стека.

`pop()`: зняти елемент з вершини.

`peek()` (або `top()`): подивитись на верхній елемент, не знімаючи його.

Ви не можете "дістати" елемент з середини стопки, не прибравши всі, що лежать зверху. Ця простота та обмеженість є надзвичайно корисною для моделювання певних процесів, наприклад, відстеження викликів функцій у програмі або реалізації кнопки "Назад" у браузері.

Черга: Принцип FIFO (First-In, First-Out)

Тепер уявіть чергу людей до каси. Нова людина стає в кінець черги, а обслуговують того, хто прийшов першим і стоїть на початку. Це принцип FIFO — "першим прийшов, першим пішов".

АТД "Черга" надає такий інтерфейс:

`enqueue(item)`: додати елемент в кінець черги.

`dequeue()`: видалити елемент з початку черги.

`front()`: подивитись на перший елемент у черзі.

Черги незамінні, коли потрібно обробляти задачі в порядку їх надходження: завдання на друк для принтера, обробка мережевих пакетів, алгоритми пошуку в ширину.

Що чекає на вас у цьому розділі?

Ми заклали теоретичний фундамент. Ми побачили дві фундаментально різні філософії зберігання послідовностей (суцільну та зв'язну) та дві важливі поведінкові абстракції (LIFO та FIFO).

Далі в цьому розділі ми зануримось у деталі кожної з цих чотирьох структур. Ми:

- Розглянемо їхні класичні реалізації.
- Напишемо псевдокод для всіх основних операцій.
- Проаналізуємо їхню продуктивність та вимоги до пам'яті.
- Розв'яжемо практичні задачі, щоб ви навчились відчувати, коли і який інструмент є найкращим для роботи.

Опанування цих базових будівельних блоків є першим і найважливішим кроком на шляху до того, щоб стати справжнім архітектором ефективних програмних рішень.

2.1. Масив: статичний та динамічний

1. Теоретичні відомості

1.1. Статичний масив: основа основ

У попередньому вступі ми говорили про дві філософії зберігання даних: суцільну та зв'язну. Статичний масив — це найчистіше втілення першої філософії.

Ідея: Уявіть собі ряд пронумерованих паркувальних місць або полицю для книг з чітко розміченими секціями. Ключова властивість — всі елементи знаходяться в пам'яті в одному суцільному (неперервному) блоці, один за одним. Кожен елемент має свій унікальний номер — індекс, зазвичай починаючи з 0.

Представлення в пам'яті: Коли ви оголошуєте в програмі масив, наприклад, на 10 цілих чисел, компілятор виділяє в оперативній пам'яті єдиний блок,

достатній для зберігання рівно 10 чисел вашого типу. Розмір цього блоку фіксується на етапі компіляції і не може бути змінений під час виконання програми.

Магія доступу за $O(1)$: Головна перевага суцільного зберігання — надзвичайно швидкий довільний доступ до будь-якого елемента. Щоб знайти елемент `A[i]`, комп'ютеру не потрібно переглядати масив. Він миттєво обчислює його точну адресу в пам'яті за простою формулою:

Адреса(A[i]) = (Початкова адреса масиву A) + i × (розмір одного елемента)

Пояснення формули:

- Початкова адреса масиву A: Адреса першого байта виділеного блоку пам'яті.
- *i*: Індекс елемента, до якого ми хочемо звернутися.
- розмір одного елемента: Кількість байтів, що займає один елемент (напр., 4 байти для `int`).

Оскільки це проста арифметична операція, вона виконується за константний час, незалежно від розміру масиву. Саме це позначається як $O(1)$.

Операції над статичним масивом:

- Доступ за індексом: Отримання або зміна значення елемента.

```
ФУНКЦІЯ GetElement(Array A, index):
    // Перевіряємо, чи індекс не виходить за межі масиву
    ЯКЩО index >= 0 І index < довжина(A) ТО
        ПОВЕРНУТИ A[index]
    ІНАКШЕ
        ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Індекс поза межами масиву")
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$

- Пошук елемента за значенням:

```
ФУНКЦІЯ LinearSearch(Array A, key):
    ДЛЯ і ВІД 0 ДО довжина(A)-1:
        ЯКЩО A[i] == key ТО
            ПОВЕРНУТИ і // Повертаємо індекс знайденого елемента
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
    ПОВЕРНУТИ -1 // Елемент не знайдено
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(n)$, оскільки в найгіршому випадку доведеться переглянути весь масив.

– Вставка та видалення: Ці операції не є природними для статичного масиву, оскільки його розмір незмінний. Можна лише перезаписати значення існуючого елемента.

1.2. Динамічний масив (Вектор): ілюзія нескінченності

Що робити, якщо ми не знаємо наперед, скільки елементів нам знадобиться зберігати? Статичний масив тут не підходить. Рішенням є динамічний масив (в C++ відомий як `std::vector`, в Java — `ArrayList`, в Python — `list`).

Ідея: Динамічний масив — це розумна "обгортка" над звичайним статичним масивом, яка приховує від програміста управління пам'яттю. Він створює ілюзію, що масив може нескінченно рости.

Як це працює? Всередині динамічний масив оперує двома ключовими поняттями:

– `size` (розмір): Фактична кількість елементів, що наразі зберігаються в масиві.

– `capacity` (місткість): Розмір внутрішнього статичного масиву, який був виділений "із запасом".

Завжди виконується умова: `size` $\leq$ `capacity`.

Операція додавання в кінець (`append`):

– Сценарій 1 (простий): Якщо `size < capacity`, тобто у внутрішньому масиві є вільне місце.

1. Новий елемент додається в комірку з індексом `size`.
2. Значення `size` збільшується на 1. Ця операція має складність $O(1)$.

– Сценарій 2 (дорогий): Якщо `size == capacity`, і вільного місця більше немає.

3. Перерозподіл (Reallocation): Створюється новий, більший статичний масив. Зазвичай його місткість вдвічі більша за стару (`new_capacity = capacity * 2`).
4. Копіювання: Всі елементи зі старого масиву копіюються в новий. Це операція складності $O(n)$, де n — поточний розмір.

5. Звільнення: Старий масив видаляється з пам’яті.
6. Вставка: Новий елемент додається в кінець нового масиву.
7. `size` та `capacity` оновлюються.

Амортизаційний аналіз: Хоча операція перерозподілу є дуже дорогою ($O(n)$), вона трапляється рідко. Якщо ми подвоюємо місткість щоразу, то вартість копіювання, “розмазана” на велику кількість дешевих $O(1)$ вставок, в середньому дає константну складність. Тому кажуть, що операція додавання в динамічний масив має амортизаційну складність $O(1)$.

Псевдокод операцій динамічного масиву:

```
// Структура, що описує динамічний масив
Структура DynamicArray:
    data      // Вказівник на внутрішній статичний масив
    size      // Кількість елементів
    capacity  // Поточна місткість

// Додавання елемента в кінець
ФУНКЦІЯ Append(DA, item):
    // Перевіряємо, чи є вільне місце
    ЯКЩО DA.size == DA.capacity ТО
        // Немає місця, подвоюємо місткість
        new_capacity = DA.capacity * 2
        new_data = новий_масив(new_capacity)
        // Копіюємо старі дані в новий масив
        ДЛЯ i ВІД 0 ДО DA.size-1:
            new_data[i] = DA.data[i]
        КІНЕЦЬ ДЛЯ
        // Звільняємо стару пам'ять і оновлюємо вказівники
        ВИДАЛИТИ DA.data
        DA.data = new_data
        DA.capacity = new_capacity
    КІНЕЦЬ ЯКЩО

    // Додаємо новий елемент
    DA.data[DA.size] = item
    DA.size = DA.size + 1
КІНЕЦЬ ФУНКЦІЇ

// Вставка елемента за індексом
ФУНКЦІЯ InsertAt(DA, index, item):
    // (Перевірка наявності місця, як в Append, тут пропущена для про
стоти)
    // Зсуваємо всі елементи праворуч від індексу
```

```
    ДЛЯ i ВІД DA.size ДО index+1 кроком -1:
        DA.data[i] = DA.data[i-1]
    КІНЕЦЬ ДЛЯ
    // Вставляємо новий елемент
    DA.data[index] = item
    DA.size = DA.size + 1
КІНЕЦЬ ФУНКЦІЇ // Складність: O(n)

// Видалення елемента за індексом
ФУНКЦІЯ RemoveAt(DA, index):
    // Зсуваємо всі елементи ліворуч, затираючи елемент за індексом
    ДЛЯ i ВІД index ДО DA.size-2:
        DA.data[i] = DA.data[i+1]
    КІНЕЦЬ ДЛЯ
    DA.size = DA.size - 1
КІНЕЦЬ ФУНКЦІЇ // Складність: O(n)
```

2. Область застосування

- Статичні масиви: Використовуються в низькорівневому програмуванні, вбудованих системах, при роботі з графікою (буфери зображень), коли розмір даних відомий наперед і критична кожна наносекунда продуктивності.
- Динамічні масиви: Є стандартною реалізацією списків у більшості високорівневих мов програмування (`std::vector` в C++, `ArrayList` в Java, `list` в Python). Вони є "робочою конячкою" і першим вибором для зберігання послідовностей у 90% випадків.

3. Приклади застосування

- Масив пікселів: Зображення на екрані — це двовимірний масив пікселів.
- Буфер даних: При читанні файлу або отриманні даних з мережі інформація часто спершу завантажується у масив-буфер фіксованого розміру.
- Список користувачів: Список зареєстрованих користувачів у веб-додатку, швидше за все, буде зберігатися у структурі, що поводиться як динамічний масив.

Переваги та недоліки

	Переваги	Недоліки
Статичний масив	Дуже швидкий довільний доступ	Фіксований розмір, негнучкість. Немож

	Переваги	Недоліки
	($O(1)$). Ефективність по пам’яті (немає додаткових витрат).	ливість вставки/видалення.
Динамічний масив	Гнучкий розмір. Швидкий довільний доступ ($O(1)$). Швидке додавання в кінець (амортизований $O(1)$).	Повільна вставка/видалення в середині ($O(n)$). Можливе марнування пам’яті (`capacity` > `size`). Рідкісні, але дорогі операції перерозподілу.

Контрольні запитання

1. Поясніть, чому доступ до елемента масиву за індексом має складність $O(1)$. Напишіть формулу обчислення адреси.
2. У чому полягає ключова відмінність між статичним та динамічним масивом?
3. Що таке `size` та `capacity` динамічного масиву? Чому `capacity` зазвичай більше за `size`?
4. Поясніть своїми словами, що таке амортизаційний аналіз на прикладі операції додавання елемента в кінець динамічного масиву.
5. Чому вставка елемента в середину динамічного масиву є операцією зі складністю $O(n)$?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація власного динамічного масиву (Вектора).

2. Мета

Набути глибокого розуміння внутрішньої роботи динамічних масивів шляхом створення власної реалізації з управлінням пам’яттю, місткістю та розміром.

3. Завдання

Створити клас `DynamicArray`, який імітує поведінку `std::vector` або `ArrayList`. Клас має надавати методи для додавання елементів, доступу за індексом та отримання поточного розміру і місткості.

4. Опис

Клас `DynamicArray` повинен мати три приватні поля: вказівник на внутрішній статичний масив, змінну `size` та змінну `capacity`. Початкова місткість може бути невеликою (напр., 4 елементи).

5. Порядок виконання

1. Створіть клас: Опишіть клас `DynamicArray` з полями `data`, `size`, `capacity`.
2. Реалізуйте конструктор: Конструктор має виділяти початковий масив певної місткості та ініціалізувати `size=0`.
3. Реалізуйте деструктор: Він має звільняти пам'ять, виділену для `data`.
4. Реалізуйте метод `append`: Керуючись псевдокодом, реалізуйте логіку додавання елемента в кінець, включаючи перевірку `size == capacity` та механізм перерозподілу пам'яті (створення нового масиву, копіювання, видалення старого).
5. Реалізуйте доступ: Додайте методи `get(index)` та `set(index, value)` для доступу до елементів. Не забудьте про перевірку виходу за межі масиву.
6. Створіть тест: Напишіть функцію `main`, яка створює об'єкт вашого класу, додає до нього більше елементів, ніж початкова місткість, щоб спрацював механізм перерозподілу, та виводить вміст на екран.

6. Результат

Повністю функціональний клас, що імітує поведінку динамічного масиву.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією класу `DynamicArray`.
2. Продемонструвати роботу додатку: показати, як змінюються `size` та `capacity` при додаванні елементів, особливо в момент перерозподілу пам'яті.
3. Бути готовим відповісти на питання:
    - Пояснити, як працює ваш механізм перерозподілу.
    - Чому подвоєння місткості є хорошою стратегією?

- Яка складність кожної з реалізованих вами операцій?

Тести

1. Яка операція є найшвидшою для масиву з N елементів?

a) Пошук елемента за значенням.

b) Видалення елемента з середини.

c) Доступ до елемента за індексом.

d) Вставка елемента на початок.

2. Амортизаційна складність $O(1)$ для додавання елемента в кінець динамічного масиву означає, що:

a) Кожна операція додавання виконується за однаковий час.

b) У найгіршому випадку операція виконується за $O(1)$.

c) Середня вартість операції, “розмазана” на велику кількість додавань, є константною.

d) Динамічний масив ніколи не змінює свій розмір.

3. Якщо динамічний масив має `size = 8` та `capacity = 8`, що відбудеться при наступному додаванні елемента?

a) Програма видасть помилку.

b) Елемент буде додано, `size` стане 9.

c) Відбудеться перерозподіл: буде створено новий масив більшої місткості.

d) Останній елемент буде перезаписано.

4. Для якої з наступних задач масив є найбільш підходящою структурою даних?

a) Зберігання історії редагувань у текстовому редакторі для функції “Undo”.

b) Представлення зображення у вигляді сітки пікселів.

c) Управління чергою завдань для друку.

d) Моделювання соціальної мережі.

5. Головний недолік динамічного масиву порівняно зі зв’язним списком — це:

a) Повільний довільний доступ за індексом.

b) Повільна ітерація по всіх елементах.

c) Повільна вставка та видалення елементів у середині списку.

d) Великі витрати пам’яті на вказівники.

Відповіді: 1-c, 2-c, 3-c, 4-b, 5-c

Тести підвищеної складності

1. Розглянемо динамічний масив, який при заповненні збільшує свою місткість у 1.5 рази замість стандартного подвоєння. Як це вплине на амортизаційну складність операції додавання в кінець?

a) Асимптотична складність покращиться до O(logn), оскільки масив зростає повільніше.

b) Асимптотична складність погіршиться до O(n), оскільки операції перерозподілу будуть відбуватися частіше.

c) Амортизаційна складність залишиться O(1), але прихована константа зросте, оскільки вартість копіювання буде "розмазуватись" на меншу кількість дешевих операцій.

d) Це призведе до помилки, оскільки місткість має зростати лише на ціле число.

2. У мові програмування, в яких двовимірні масиви зберігаються по рядках, який з двох циклів, що обчислюють суму елементів матриці M[N][N], буде значно швидшим на практиці і чому?

a) Цикл for j from 0 to N-1: for i from 0 to N-1: sum += M[i][j] буде швидшим через кращу часову локальність.

b) Цикл for i from 0 to N-1: for j from 0 to N-1: sum += M[i][j] буде швидшим через кращу просторову локальність та ефективніше використання кешу процесора.

c) Обидва цикли будуть працювати з абсолютно однаковою швидкістю, оскільки виконують однакову кількість операцій.

d) Цикл for j from 0 to N-1: for i from 0 to N-1: sum += M[i][j] буде швидшим, оскільки доступ по стовпцях завжди ефективніший.

3. Вам дано двовимірний масив int A[10][20], де int займає 4 байти. Якщо початкова адреса масиву (адреса елемента A[0][0]) дорівнює 1000, яка буде адреса елемента A[5][10] за умови зберігання по рядках?

a) 1440

b) 1620

c) 2200

d) 1150

4. Ви проєктуєте систему, яка зберігає історію транзакцій. Найчастіші операції — додавання нової транзакції на початок списку та перегляд 10 останніх транзакцій. Чому динамічний масив (вектор) є поганим вибором для цього завдання?

a) Тому що доступ до елемента за індексом в динамічному масиві має складність O(n).

b) Тому що динамічний масив не може зберігати об'єкти транзакцій.

c) Тому що кожна операція додавання на початок списку вимагатиме зсуву всіх існуючих елементів, що призведе до складності O(n).

d) Тому що динамічний масив не може змінювати свій розмір.

5. У чому полягає ключова відмінність у представленні в пам'яті між "зубчастим масивом" та стандартним двовимірним масивом M[R][C]?

a) У зубчастому масиві всі рядки гарантовано мають однакову довжину.

b) Стандартний двовимірний масив зберігається як один суцільний блок пам'яті, тоді як зубчастий масив — це масив вказівників, кожен з яких вказує на окремий одновимірний масив-рядок.

c) Доступ до елемента M[i][j] у зубчастому масиві швидший, ніж у стандартному.

d) Зубчасті масиви не можуть бути динамічними.

Відповіді

c) Амортизаційна складність O(1) зберігається для будь-якого коефіцієнта росту > 1. Однак, чим менший коефіцієнт, тим частіше відбуваються дорогі операції перерозподілу, що збільшує усереднену ("амортизовану") вартість однієї операції.

b) При зберіганні по рядках елементи M[i][0], M[i][1], M[i][2], ... знаходяться в пам'яті послідовно. Цикл for i ... for j ... читає пам'ять лінійно, що максимально ефективно використовує кеш-лінії процесора (просторова локальність). Другий варіант циклу "стрибає" по пам'яті, що призводить до великої кількості кеш-промахів.

a) Формула для адреси в row-major order: base_address + (row_index * num_cols + col_index) * element_size. Підставляємо значення: 1000 + (5 * 20 + 10) * 4 = 1000 + (100 + 10) * 4 = 1000 + 110 * 4 = 1000 + 440 = 1440.

c) Динамічний масив оптимізований для додавання в кінець. Додавання на початок — це найгірший сценарій, який вимагає зсуву всіх n елементів, що робить операцію дуже повільною (O(n)). Для такого завдання набагато краще підійшов би двозв'язний список.

b) Це фундаментальна відмінність. Двовимірний масив R×C — це єдиний блок пам'яті розміром R·C·sizeof(type). Зубчастий масив — це масив вказівників, кожен з яких вказує на потенційно різний за розміром та місцем у пам'яті масив-рядок.

Завдання для самостійної роботи

1. Реалізація видалення. Додайте до вашого класу `DynamicArray` метод `removeAt(index)`, який видаляє елемент за вказаним індексом, зсовуючи всі наступні елементи ліворуч.
2. Зменшення місткості. Модифікуйте ваш `DynamicArray` так, щоб він не тільки збільшувався, але й зменшувався. Наприклад, якщо `size` стає меншим за `capacity / 4`, зменшуйте місткість вдвічі.
3. Аналіз коефіцієнта росту. Дослідіть, як зміниться амортизаційна вартість додавання, якщо збільшувати місткість не вдвічі, а в 1.5 рази або додавати фіксовану кількість комірок (напр., +100).
4. Двовимірний масив. Напишіть клас для роботи з двовимірним динамічним масивом (матрицею), який дозволяє додавати та видаляти рядки.
5. Пошук у відсортованому масиві. Реалізуйте бінарний пошук для вашого класу `DynamicArray`.
6. Перевертання масиву. Напишіть функцію, що перевертає масив “на місці” (in-place), тобто не використовуючи додаткового масиву. Яка її складність?
7. Циклічний зсув. Реалізуйте функцію, що виконує циклічний зсув елементів масиву на *k* позицій вправо. Наприклад, `{1,2,3,4,5}` з k=2 стає `{4,5,1,2,3}`. Спробуйте досягти складності $O(n)$.
8. Пошук дублікатів. Напишіть ефективний алгоритм для перевірки, чи містить масив дублікати. Проаналізуйте його складність.

9. Об’єднання відсортованих масивів. Напишіть функцію, що приймає два відсортовані масиви і повертає новий відсортований масив, що містить елементи з обох.
10. Обертання матриці. Напишіть функцію, що обертає квадратну матрицю $N \times N$ на 90 градусів за годинниковою стрілкою.
11. Спіральний обхід. Реалізуйте функцію, що виводить елементи двовимірної матриці по спіралі.
12. Максимальний підмасив. Знайдіть неперервний підмасив у масиві цілих чисел (можуть бути від’ємні), сума елементів якого є максимальною (це завдання можна розв’язати за $O(n)$).
13. Реалізація `insertAt`. Додайте до вашого класу `DynamicArray` метод `insertAt(index, item)`, що вставляє елемент за вказаним індексом.
14. Есе-порівняння. Напишіть есе, що порівнює `std::vector` (C++) та `list` (Python). Які спільні ідеї лежать в їх основі, і які є ключові відмінності в їх поведінці та гарантіях?
15. Аналіз реальних реалізацій. Дослідіть вихідний код реалізації `ArrayList` в OpenJDK або `std::vector` в libc++. Знайдіть код, що відповідає за перерозподіл пам’яті.

2.2. Рядок.

1. Теоретичні відомості

1.1. Рядок як послідовність символів

На базовому рівні, рядок — це просто послідовність символів. “Hello, world!”, ваше ім’я, адреса веб-сайту — все це рядки. Найпростіший спосіб представити таку послідовність у пам’яті комп’ютера — це використати структуру, яку ми вже знаємо: масив. У цьому випадку, це буде масив символів (`char`).

Цей підхід, де рядок є просто масивом символів, що закінчується спеціальним нуль-символом (`\0`), довгий час був стандартом у таких мовах, як C. Однак, він має суттєві недоліки, пов’язані з безпекою та ефективністю модифікацій. Сучасні мови програмування використовують більш просунуті та абстрактні підходи до реалізації рядків. Давайте розглянемо чотири основні концепції.

1.2. Незмінний рядок (Immutable String): Простота та Безпека

Ідея: Уявіть, що рядок — це текст, вигравіруваний на камені. Щойно ви його створили, ви вже не можете його змінити. Будь-яка “модифікація” насправді створює абсолютно новий камінь з новим текстом.

Це і є принцип незмінності (immutability). Рядок, створений один раз, більше ніколи не змінюється. Цей підхід є стандартним у таких мовах, як Java, Python, C#.

Представлення в пам’яті: Зазвичай це масив символів, розмір якого відомий і зберігається окремо.

Операції та їх складність:

- Довжина (`length()`): Оскільки довжина зберігається в окремому полі, отримання її є миттєвим. *Складність: $O(1)$*
- Отримання символу за індексом (`charAt(index)`): Оскільки в основі лежить масив, доступ за індексом також миттєвий. *Складність: $O(1)$*
- Конкатенація (об’єднання) двох рядків (`concatenate(other_string)`): Це найцікавіша операція. Оскільки ми не можемо змінити існуючі рядки, ми змушені створювати третій, абсолютно новий.

```
ФУНКЦІЯ Concatenate(string1, string2):
    len1 = string1.length()
    len2 = string2.length()

    // 1. Виділяємо пам'ять для нового рядка
    new_array = новий_масив_символів(len1 + len2)

    // 2. Копіюємо символи з першого рядка
    ДЛЯ i ВІД 0 ДО len1-1:
        new_array[i] = string1.charAt(i)
    КІНЕЦЬ ДЛЯ

    // 3. Копіюємо символи з другого рядка
    ДЛЯ i ВІД 0 ДО len2-1:
        new_array[len1 + i] = string2.charAt(i)
    КІНЕЦЬ ДЛЯ

    // Створюємо та повертаємо новий об'єкт рядка
    ПОВЕРНУТИ new String(new_array)
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(n+m)$, де n та m — довжини рядків, оскільки ми змушені копіювати всі символи.

– Отримання підрядка (`substring(start, end)`): Залежить від реалізації. “Наївна” реалізація створить новий масив і скопіює туди символи, що дасть складність $O(k)$, де k — довжина підрядка. Більш просунуті реалізації можуть створити новий об’єкт рядка, який просто посилається на той самий масив символів, що й оригінал, але має інші вказівники на початок та кінець. Це робить операцію дуже швидкою, $O(1)$.

1.3. Змінний буфер (Mutable Buffer): Ефективні модифікації

Ідея: Якщо незмінний рядок — це гравіювання на камені, то змінний буфер — це дошка, на якій можна писати крейдою, стирати та дописувати. Він створений спеціально для ефективної та багаторазової модифікації.

Представлення в пам’яті: По суті, змінний буфер — це динамічний масив, який ми вже розглядали. Він має внутрішній масив символів (`capacity`), який зазвичай більший за поточну довжину рядка (`size`).

Операції та їх складність:

– Додавання в кінець (`append(char)` або `append(string)`): Це найефективніша операція. Поки `size < capacity`, ми просто додаємо символи в кінець і збільшуємо `size`. Коли місце закінчується, відбувається перерозподіл пам’яті (створення нового, більшого масиву та копіювання даних).

Складність: Амортизована $O(1)$ *для одного символу,* $O(k)$ *для рядка довжиною* k.

```
// Дуже схоже на реалізацію для динамічного масиву
ФУНКЦІЯ Append(buffer, text_to_append):
    // Перевіряємо, чи достатньо місця, і якщо ні, збільшуємо ca
pacity
    EnsureCapacity(buffer, довжина(text_to_append))

    // Копіюємо символи text_to_append в кінець буфера
    // ... логіка копіювання ...

    // Оновлюємо size
    buffer.size = buffer.size + довжина(text_to_append)
КІНЕЦЬ ФУНКЦІЇ
```

– Вставка в середину (`insert(index, string)`): Ця операція вимагає зсуву всіх символів, що знаходяться праворуч від `index`, щоб звільнити місце.

  Складність: $O(n + k)$, де n — поточна довжина буфера, а k — довжина рядка, що вставляється.

– Видалення (`delete(start, end)`): Аналогічно вимагає зсуву символів, щоб "заповнити" прогалину.

  Складність: $O(n)$.

1.4. Просунуті структури: Мотузка (Rope) та Таблиця частин (Piece Table)

Для специфічних завдань, як-от робота текстових редакторів, були створені ще більш складні та ефективні структури.

Мотузка (Rope)

Ідея: Замість зберігання всіх символів в одному масиві, представити довгий рядок у вигляді бінарного дерева. Листки цього дерева містять короткі, незмінні підрядки, а внутрішні вузли зберігають інформацію про структуру (наприклад, сумарну довжину символів у лівому піддереві).

Операції та їх складність:

– Конкатенація: Це найсильніша сторона мотузки. Щоб об'єднати два рядки, представлені як мотузки `rope1` та `rope2`, достатньо створити новий кореневий вузол, у якого `rope1` буде лівим нащадком, а `rope2` — правим.

  Складність: $O(1)$ (надзвичайно швидко).

– Отримання символу за індексом (`charAt(index)`): Потрібно спускатися по дереву від кореня, на кожному кроці порівнюючи індекс з довжиною лівого піддерева, щоб зрозуміти, в яку гілку йти.

  Складність: $O(\log n)$ (швидше, ніж у зв'язного списку, але повільніше, ніж у масиву).

– Вставка/Видалення: Зазвичай зводиться до операції "розрізання" мотузки на дві частини та їх подальшого об'єднання з новим рядком.

  Складність: $O(\log n)$.

Таблиця частин (Piece Table)

Ідея: Це структура даних, що є стандартом де-факто для професійних текстових редакторів (VS Code, Atom). Її головний принцип: оригінальний текст ніколи не змінюється. Представлення:

1. Original Buffer: Незмінний масив, що містить початковий текст файлу.
2. Add Buffer: Змінний буфер, куди дописуються всі нові символи, які вводить користувач.
3. Piece Table: Головна структура. Це список (або дерево) “шматків” (pieces). Кожен шматок — це об’єкт, що містить вказівник на один з буферів (original або add), початкову позицію та довжину.

Як це працює:

– Початковий стан: Є лише один шматок, що вказує на весь Original Buffer.

– Вставка: Коли користувач вставляє текст, цей текст додається в кінець Add Buffer. Поточний шматок розрізається на два, а між ними вставляється новий шматок, що вказує на щойно доданий фрагмент в Add Buffer.

– Видалення: Видалення тексту — це просто операція над списком шматків: один шматок може бути розрізаний, інший — скорочений або видалений. Самі символи в буферах не зачіпаються.

2. Область застосування

– Незмінні рядки: Основа більшості сучасних мов програмування (Java, Python, C#, JavaScript). Ідеальні для передачі даних, використання як ключів у словниках, забезпечення безпеки у багатопотокових програмах.

– Змінні буфери: Використовуються для ефективного конструювання рядків з багатьох частин, наприклад, при формуванні HTML-сторінки або SQL-запиту в циклі. (`StringBuilder` в Java/C#, `io.StringIO` в Python).

– Мотузка (Rope) та Таблиця частин (Piece Table): Використовуються у системах, що працюють з дуже великими текстами і вимагають частих модифікацій та миттєвої функції Undo/Redo, — текстові редактори, текстові процесори.

3. Приклади застосування

– `"a" + "b" + "c"` в Java: Цей код створить декілька проміжних незмінних об’єктів рядків (`"ab"`), що є неефективним.

– `new StringBuilder().append("a").append("b").append("c")`: Цей код значно ефективніший, оскільки всі модифікації відбуваються в одному буфері пам’яті.

– Microsoft Visual Studio Code: Використовує Piece Table як основну структуру для представлення текстових файлів, що забезпечує високу продуктивність при редагуванні великих файлів та необмежену історію змін.

Переваги та недоліки

Структура	Переваги	Недоліки
Незмінний рядок	Безпечність (thread-safe), простота, надійність, можливість кешування хеш-коду.	Дуже повільна конкатенація та будь-які модифікації.
Змінний буфер	Надзвичайно швидке додавання в кінець (конкатенація).	Повільні вставка та видалення в середині. Не є потокобезпечним.
Мотузка (Rope)	Дуже швидка конкатенація ($O(1)$). Швидкі вставка/видалення ($O(\log n)$).	Повільніший доступ за індексом ($O(\log n)$). Більша складність реалізації та витрати пам’яті.
Таблиця частин	Майже миттєві вставка/видалення. Необмежена історія змін (Undo/Redo).	Повільне отримання всього тексту (потрібно збирати зі шматків). Висока складність реалізації.

Контрольні запитання

1. У чому полягає основний принцип незмінних (immutable) рядків?
2. Поясніть, чому конкатенація незмінних рядків у циклі є поганою ідеєю з точки зору продуктивності.

3. Опишіть, як працює механізм перерозподілу пам’яті у змінному буфері (динамічному масиві).
4. Для яких операцій Мотузка (Rope) є значно ефективнішою за звичайний рядок на основі масиву?
5. Чому Таблиця частин (Piece Table) є ідеальним вибором для текстових редакторів?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація власного змінного буфера рядків (StringBuilder).

3. Мета

Набути глибокого розуміння внутрішньої роботи змінних рядків шляхом створення власної реалізації з управлінням пам’яттю, місткістю та розміром.

4. Завдання

Створити клас **`MutableStringBuffer`**, який імітує поведінку **`StringBuilder`**. Клас має надавати методи для додавання символів та інших рядків, а також метод для отримання фінального рядка.

4. Опис

Клас **`MutableStringBuffer`** повинен інкапсулювати логіку динамічного масиву символів. Він повинен мати приватні поля: вказівник на внутрішній масив, змінну **`size`** та змінну **`capacity`**.

6. Порядок виконання
    1. Створіть клас: Опишіть клас **`MutableStringBuffer`** з необхідними полями.
    2. Реалізуйте конструктор: Конструктор має виділяти початковий масив певної місткості (напр., 16 символів) та ініціалізувати **`size=0`**.
    3. Реалізуйте деструктор: Він має звільняти пам’ять, виділену для внутрішнього масиву.
    4. Реалізуйте метод **`append(string)`**: Керуючись псевдокодом для динамічного масиву, реалізуйте логіку додавання рядка в кінець. Це включає перевірку **`size + string.length() > capacity`** та механізм перерозподілу пам’яті.

5. Реалізуйте метод **`toString()`**: Цей метод має створювати та повертати новий незмінний рядок на основі поточного вмісту буфера.
6. Створіть тест: Напишіть функцію **`main`**, яка створює об'єкт вашого класу, додає до нього декілька рядків у циклі (щоб гарантовано викликати перерозподіл) та виводить фінальний результат.

7. Результат

Повністю функціональний клас, що дозволяє ефективно будувати рядки з частин.

8. Після виконання лабораторної роботи студент повинен надати:
    - Вихідний код програми з реалізацією класу **`MutableStringBuffer`**.
    - Продемонструвати роботу додатку: показати, як змінюється **`size`** та **`capacity`** при додаванні даних.
9. Бути готовим відповісти на питання:
    - Пояснити, як працює ваш механізм перерозподілу пам'яті.
    - Чому ваш клас ефективніший за просту конкатенацію незмінних рядків у циклі?
    - Яка складність кожної з реалізованих вами операцій?

7. Тести

1. Яка часова складність операції **`s1 + s2`**, де **`s1`** та **`s2`** — незмінні рядки довжиною N та M відповідно?

a) $O(1)$

b) $O(N)$

c) $O(N + M)$

d) $O(N \cdot M)$

2. Яка структура даних лежить в основі реалізації **`StringBuilder`** або **`ArrayList`**?

a) Зв'язаний список

b) Динамічний масив

c) Хеш-таблиця

d) Бінарне дерево

3. Незмінність (immutability) рядків є важливою для:

a) Швидкої конкатенації.

b) Безпечної роботи у багатопотоковому середовищі.

c) Швидкої вставки символів у середину рядка.

d) Зменшення використання пам’яті.

4. Яка структура даних забезпечує конкатенацію двох довгих рядків за час $O(1)$?

a) Незмінний рядок

b) Змінний буфер

c) Мотузка (Rope)

d) Масив

5. Для якого завдання Таблиця частин (Piece Table) є найбільш оптимальною структурою?

a) Зберігання ключів у словнику.

b) Представлення текстового файлу у високопродуктивному текстовому редакторі.

c) Управління чергою завдань на друк.

d) Зберігання матриці суміжності графа.

Відповіді: 1-c, 2-b, 3-b, 4-c, 5-b

Тести підвищеної складності

1. Розглянемо код, який будує рядок у циклі: `result = ""`; `for i in 1..N: result = result + "a"`. Якщо в мові використовуються незмінні (immutable) рядки, яка буде фінальна часова складність цієї операції?

a) $O(N)$, оскільки цикл виконується N разів, і кожна операція додавання є $O(1)$.

b) $O(N \log N)$, оскільки на кожному кроці створюється новий об’єкт, що схоже на логіку сортування злиттям.

c) $O(N^2)$, оскільки на k-тій ітерації створюється новий рядок довжиною k, що вимагає копіювання *k-1* старих символів. Сумарно це $1 + 2 + \ldots + (N - 1)$ операцій.

d) $O(\log N)$, тому що компілятор автоматично оптимізує конкатенацію.

2. Яке твердження найбільш точно описує компроміс продуктивності при використанні структури даних Мотузка порівняно зі звичайним рядком на основі масиву?

a) Мотузка швидша за масив для всіх операцій завдяки своїй деревоподібній структурі.

b) Мотузка забезпечує значно швидшу конкатенацію ($O(1)$) та вставку/видалення ($O(\log n)$), але розплачується за це повільнішим доступом до символу за індексом ($O(\log n)$ замість $O(1)$).

c) Мотузка забезпечує доступ за індексом за $O(1)$, але має повільну конкатенацію $O(n)$.

d) Мотузка використовує менше пам’яті, ніж звичайний рядок, але всі операції виконуються за лінійний час.

3. При видаленні фрагмента тексту в текстовому редакторі, що використовує структуру даних Таблиця частин, що відбувається “під капотом”?

a) Великий блок пам’яті, що містить текст, копіюється в новий блок за винятком видалених символів.

b) Символи, що видаляються, позначаються спеціальним прапорцем “видалено”, але залишаються в пам’яті.

c) Один або декілька “шматків” у таблиці модифікуються (напр., розрізаються або змінюють свою довжину), але основні масиви символів не змінюються.

d) Викликається збирач сміття для фізичного видалення символів з пам’яті.

4. У чому полягає основна перевага незмінних рядків у багатопотоковому програмуванні?

a) Вони завжди займають менше пам’яті.

b) Їх можна безпечно передавати між потоками без необхідності блокувань, оскільки їхній стан гарантовано не може бути змінений.

c) Будь-яка операція над ними є атомарною.

d) Вони дозволяють швидше виконувати операцію конкатенації.

5. Ви проєктуєте систему для спільного редагування коду в реальному часі, де багато користувачів одночасно вносять невеликі зміни. Яку структуру даних для представлення тексту доцільно обрати в першу чергу?

a) Простий динамічний масив (змінний буфер), оскільки він ефективний для модифікацій.

b) Незмінний рядок, оскільки він гарантує безпеку даних.

c) Мотузку або Таблицю частин, оскільки вони ефективно обробляють часті невеликі вставки/видалення та дозволяють легко реалізувати історію версій.

d) Хеш-таблицю, де ключем є номер рядка, а значенням — текст рядка.

Відповіді

1. c) Це класичний приклад, що демонструє неефективність конкатенації незмінних рядків. На кожній ітерації створюється новий об’єкт і копіюються всі символи з попереднього, що призводить до квадратичної складності.

2. b) Це ключовий компроміс Мотузки: вона перетворює дорогу операцію конкатенації на надзвичайно дешеву, але жертвує миттєвим доступом за індексом, роблячи його логарифмічним.

3. c) Таблиця частин працює, маніпулюючи дескрипторами (“шматками”), які вказують на фрагменти тексту. Модифікація тексту — це, по суті, модифікація цього списку дескрипторів, а не самих текстових даних.

4. b) Оскільки стан незмінного об’єкта не може бути змінений після створення, його можна безпечно читати з будь-якої кількості потоків одночасно без ризику виникнення стану гонитви і без потреби у складних механізмах синхронізації.

5. c) Ці структури даних були розроблені саме для таких завдань. Вони дозволяють швидко вносити зміни в середину великого документа і зберігати його попередні версії практично безкоштовно, що є критичним для спільного редагування та функції undo/redo.

Завдання для самостійної роботи

1. Аналіз продуктивності. Напишіть програму, яка емпірично порівнює швидкість побудови довгого рядка (100 000+ символів) двома способами: за допомогою конкатенації незмінних рядків у циклі та за допомогою змінного буфера.

2. Реалізація **`insert`**. Додайте до вашого класу **`MutableStringBuffer`** метод **`insert(index, string)`**, що вставляє рядок у вказану позицію.
3. Реалізація **`delete`**. Додайте до вашого класу **`MutableStringBuffer`** метод **`delete(start, end)`**, що видаляє підрядок.
4. Простий Rope. Спробуйте реалізувати спрощену структуру Rope, яка підтримує лише операцію конкатенації.
5. Індекс в Rope. Опишіть детальний алгоритм для операції **`charAt(index)`** для структури Rope.
6. “String Interning”. Дослідіть, що таке “інтернування рядків” (string interning) в таких мовах, як Java або Python, і як воно допомагає економити пам’ять.
7. Порівняння реалізацій. Напишіть есе, що порівнює **`std::string`** (C++) та **`String`** (Java). Які фундаментальні відмінності в їхній ідеології та реалізації?
8. Undo/Redo з Piece Table. Опишіть, як структура Piece Table дозволяє реалізувати необмежену та миттєву історію змін (Undo/Redo).
9. Аналіз пам’яті. Оцініть, скільки пам’яті (приблизно) буде використано для зберігання 1000 версій документа по 1МБ кожна, якщо на кожному кроці змінюється лише 1 байт, при використанні: а) масиву символів (кожна версія — повна копія), б) персистентної структури даних (напр., Piece Table).
10. Вибір структури. Для кожного з наступних завдань оберіть найбільш підходящу реалізацію рядка (незмінний, буфер, rope/piece table) і обґрунтуйте свій вибір:
    a) Зберігання URL-адрес в конфігураційному файлі.
    b) Формування великого JSON-об’єкта з тисяч записів бази даних.
    c) Розробка спільного редактора документів, як Google Docs.

Зв’язаний список

Друга фундаментальна концепція зберігання даних — зв'язна. Цей розділ присвячений зв'язаним спискам, які є гнучкою та потужною альтернативою масивам для багатьох завдань.

Теоретичні відомості

У попередньому розділі ми детально розглянули масиви, що базуються на ідеї суцільного зберігання даних у пам'яті. Це дає нам блискавичний доступ до елемента за індексом, але створює серйозні проблеми, коли ми хочемо вставити або видалити елемент із середини послідовності — доводиться переміщувати безліч інших елементів.

А що, якщо відмовитись від ідеї зберігати все разом? Уявіть собі "полювання за скарбами": ви знаходите першу записку, в якій сказано, де шукати другу; у другій записці — вказівка на третю, і так далі. Самі записки можуть бути розкидані де завгодно, але разом вони утворюють чіткий послідовний маршрут.

Це і є центральна ідея зв'язного списку (Linked List). Замість одного великого блоку пам'яті, ми використовуємо безліч маленьких незалежних блоків — вузлів (nodes), які "знають" один про одного.

Будівельний блок: Вузол (Node) Кожен вузол у зв'язному списку — це проста структура, що зберігає дві речі:

1. Дані: Корисна інформація, яку ми хочемо зберігати (число, рядок, об'єкт).
2. Вказівник: Адреса наступного вузла в ланцюжку.

Весь список управляється лише одним вказівником — `head` (голова), який вказує на найперший вузол. Останній вузол у списку вказує на `NULL`, сигналізуючи про кінець ланцюжка.

Однозв'язний список

Це найпростіша форма зв'язного списку, де кожен вузол має лише один вказівник — на наступний елемент. Рухатись по такому списку можна тільки в одному напрямку.

Структура вузла:

```
Структура Node:
    data      // Дані, що зберігаються
    next      // Вказівник на наступний вузол
```

Операції над однозв'язним списком:

– Обхід: Щоб пройти по всіх елементах списку, ми починаємо з голови і послідовно переходимо за вказівниками `next`.

```
ФУНКЦІЯ Traverse(head):
    current = head
```

```
    ПОКИ current != NULL:
        Обробити(current.data)
        current = current.next
    КІНЕЦЬ ПОКИ
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(n)$, оскільки ми маємо відвідати кожен з n вузлів.

– Вставка на початок: Це одна з найсильніших сторін зв'язних списків.

1. Створюємо новий вузол.
2. Встановлюємо його вказівник `next` так, щоб він вказував на поточну голову списку.
3. Оновлюємо `head` так, щоб він тепер вказував на наш новий вузол.

```
ФУНКЦІЯ InsertAtHead(head, new_data):
    newNode = Створити_Вузол(new_data)
    newNode.next = head
    head = newNode
    ПОВЕРНУТИ head
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$, оскільки кількість операцій не залежить від розміру списку.

– Вставка: Якщо ми вже маємо вказівник на вузол, після якого потрібно зробити вставку, операція також є дуже швидкою.

1. Створюємо новий вузол.
2. Зберігаємо посилання, на яке вказував `node.next`.
3. Встановлюємо `newNode.next` на збережене посилання.
4. Оновлюємо `node.next` так, щоб він вказував на наш новий вузол.

```
ФУНКЦІЯ InsertAfter(node, new_data):
    ЯКЩО node == NULL ТО
        // Не можна вставити після неіснуючого вузла
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    newNode = Створити_Вузол(new_data)
    newNode.next = node.next
    node.next = newNode
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$.

– Вставка в кінець: Щоб додати елемент в кінець, нам спершу потрібно знайти цей кінець, пройшовши весь список від голови.

```
ФУНКЦІЯ InsertAtTail(head, new_data):
    newNode = Створити_Вузол(new_data)
    ЯКЩО head == NULL ТО // Якщо список порожній
        ПОВЕРНУТИ newNode
    КІНЕЦЬ ЯКЩО

    last = head
    ПОКИ last.next != NULL:
        last = last.next
    КІНЕЦЬ ПОКИ

    last.next = newNode
    ПОВЕРНУТИ head
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(n)$. (Примітка: цю операцію можна прискорити до $O(1)$, якщо окремо зберігати вказівник не тільки на голову (**head**), але і на хвіст (**tail**)).

– Видалення з початку:

```
ФУНКЦІЯ DeleteFromHead(head):
    ЯКЩО head == NULL ТО ПОВЕРНУТИ NULL КІНЕЦЬ ЯКЩО

    temp = head.next
    ВИДАЛИТИ head
    head = temp
    ПОВЕРНУТИ head
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$.

– Видалення останнього: Це слабке місце однозв'язного списку. Щоб видалити останній вузол, нам потрібно змінити вказівник **next** у передостаннього вузла на **NULL**. А щоб знайти передостанній вузол, ми змушені пройти весь список з початку.

```
ФУНКЦІЯ DeleteFromTail(head):
    // Обробка порожнього списку або списку з одного елемента
    ЯКЩО head == NULL АБО head.next == NULL ТО
        ВИДАЛИТИ head
        ПОВЕРНУТИ NULL
    КІНЕЦЬ ЯКЩО

    // Пошук передостаннього вузла
```

```
        second_last = head
        ПОКИ second_last.next.next != NULL:
            second_last = second_last.next
        КІНЕЦЬ ПОКИ

        // Видалення останнього вузла та оновлення вказівника
        ВИДАЛИТИ second_last.next
        second_last.next = NULL
        ПОВЕРНУТИ head
    КІНЕЦЬ ФУНКЦІЇ
```
Складність: $O(n)$.

– Видалення з довільного місця: Як і з видаленням останнього елемента, щоб видалити вузол `X`, нам потрібно змінити вказівник `next` у попереднього вузла. А оскільки ми не можемо рухатись назад, нам доводиться щоразу шукати цей попередній вузол, проходячи список з початку. *Складність:* $O(n)$.

Двозв'язний список (Doubly Linked List): дорога в обидва боки

Щоб вирішити проблему неефективного видалення та неможливості руху назад, був створений двозв'язний список.

Ідея: Кожен вузол тепер має два вказівники: один на наступний елемент (`next`), а інший — на попередній (`prev`).

Структура вузла:

```
Структура DNode:
    data      // Дані, що зберігаються
    next      // Вказівник на наступний вузол
    prev      // Вказівник на попередній вузол
```

Переваги:

– Можливість обходити список в обох напрямках.

– Дуже ефективне видалення вузла, якщо ми маємо на нього вказівник.

Операції над двозв'язним списком:

– Обхід: Тепер можливий у двох напрямках, якщо ми зберігаємо вказівники `head` та `tail`.

```
    ФУНКЦІЯ TraverseForward(head):
        current = head
        ПОКИ current != NULL:
            Обробити(current.data)
            current = current.next
        КІНЕЦЬ ПОКИ
```

```
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ TraverseBackward(tail):
    current = tail
    ПОКИ current != NULL:
        Обробити(current.data)
        current = current.prev
    КІНЕЦЬ ПОКИ
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(n)$.

– Вставка на початок:

```
ФУНКЦІЯ InsertAtHead(head, tail, new_data):
    newNode = Створити_Вузол(new_data)
    newNode.next = head

    ЯКЩО head != NULL ТО
        head.prev = newNode
    КІНЕЦЬ ЯКЩО

    head = newNode

    ЯКЩО tail == NULL ТО // Якщо список був порожній
        tail = newNode
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$.

– Вставка в кінець: Завдяки наявності вказівника **`tail`** ця операція також стає дуже швидкою.

```
ФУНКЦІЯ InsertAtTail(head, tail, new_data):
    newNode = Створити_Вузол(new_data)
    newNode.prev = tail

    ЯКЩО tail != NULL ТО
        tail.next = newNode
    КІНЕЦЬ ЯКЩО

    tail = newNode

    ЯКЩО head == NULL ТО // Якщо список був порожній
        head = newNode
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$.

– Вставка після певного вузла **node**: Ця операція вимагає більше маніпуляцій з вказівниками, але її суть та сама.

```
ФУНКЦІЯ InsertAfter(node, new_data):
    ЯКЩО node == NULL ТО ПОВЕРНУТИ КІНЕЦЬ ЯКЩО

    newNode = Створити_Вузол(new_data)
    newNode.next = node.next
    node.next = newNode
    newNode.prev = node

    ЯКЩО newNode.next != NULL ТО
        newNode.next.prev = newNode
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$.

– Видалення з початку:

```
ФУНКЦІЯ DeleteFromHead(head, tail):
    ЯКЩО head == NULL ТО ПОВЕРНУТИ КІНЕЦЬ ЯКЩО

    nodeToDelete = head
    head = head.next

    ЯКЩО head != NULL ТО
        head.prev = NULL
    ІНАКШЕ // Список став порожнім
        tail = NULL
    КІНЕЦЬ ЯКЩО

    ВИДАЛИТИ nodeToDelete
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$.

– Видалення останнього:

```
ФУНКЦІЯ DeleteFromTail(head, tail):
    ЯКЩО tail == NULL ТО ПОВЕРНУТИ КІНЕЦЬ ЯКЩО

    nodeToDelete = tail
    tail = tail.prev

    ЯКЩО tail != NULL ТО
        tail.next = NULL
    ІНАКШЕ // Список став порожнім
```

```
        head = NULL
    КІНЕЦЬ ЯКЩО

    ВИДАЛИТИ nodeToDelete
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$.

– Видалення довільного вузла `node`: Це головна перевага. Маючи вказівник на `node`, ми можемо легко знайти його сусідів (`node.prev` та `node.next`) і “зшити” їх між собою.

```
ФУНКЦІЯ DeleteNode(node):
    // Якщо це не голова списку, оновлюємо вказівник попередньог
о
    ЯКЩО node.prev != NULL ТО
        node.prev.next = node.next
    КІНЕЦЬ ЯКЩО

    // Якщо це не хвіст списку, оновлюємо вказівник наступного
    ЯКЩО node.next != NULL ТО
        node.next.prev = node.prev
    КІНЕЦЬ ЯКЩО

    // (Потрібно також оновити head/tail, якщо видаляється один
з них)
    ВИДАЛИТИ node
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(1)$.

Циклічний зв’язний список

Ідея: Варіація зв’язного списку (може бути як одно-, так і двозв’язним), де останній елемент вказує не на `NULL`, а назад на голову списку (`head`), утворюючи замкнене кільце.

Особливості:

– У списку немає початку і кінця у звичному розумінні.

– Обхід списку може тривати нескінченно, якщо не зупинитися, коли ми знову дійдемо до стартового вузла.

– Дозволяє легко переміщатися з кінця списку на його початок.

Коли використовувати: Циклічні списки ідеально підходять для реалізації завдань, що мають циклічну природу:

– Round-robin scheduling: Алгоритм планувальника, який по черзі надає процесорний час кожному процесу в кільцевій черзі.
– Ігрові цикли: Послідовність ходів гравців у настільній грі.
– Кільцеві буфери.

Область застосування

– Операційні системи: Списки використовуються для управління процесами, файлами, що очікують на запис, та іншими системними ресурсами.
– Веб-браузери: Кнопки “Вперед” та “Назад” реалізовані за допомогою двозв’язного списку, що зберігає історію відвіданих сторінок.
– Музичні плеєри: Плейлист, де можна перемикати треки вперед і назад, є гарним прикладом двозв’язного списку. Циклічний список може реалізовувати функцію повтору плейлиста.
– Реалізація інших структур даних: Стеки та черги часто реалізують на основі зв’язаних списків.

Приклади застосування

– Функція “Undo” в текстових редакторах: Кожна дія користувача (друк, видалення) може зберігатися як вузол у двозв’язному списку. Натискання `Ctrl+Z` переміщує нас по вказівнику `prev`, а `Ctrl+Y` — по `next`.
– Ланцюжки в хеш-таблицях: Коли в хеш-таблиці виникає колізія, елементи з однаковим хешем часто організовують у зв’язаний список.

Переваги та недоліки

Порівняємо розглянуті структури даних та масив:

Структура	Переваги	Недоліки
Масив	Дуже швидкий довільний доступ ($O(1)$). Кеш-ефективність.	Повільна вставка/видалення ($O(n)$). Фіксований розмір (для статичних).
Однозв’язний список	Швидка вставка/видалення на початку ($O(1)$).	Повільний довільний доступ та пошук ($O(n)$).

Структура	Переваги	Недоліки
	Гнучкий розмір.	Повільне видалення з кінця/середини ($O(n)$). Додаткова пам’ять на вказівники.
Двозв’язний список	Швидка вставка/видалення в будь-якому місці (якщо є вказівник на вузол) ($O(1)$). Можливість руху в обох напрямках.	Повільний довільний доступ та пошук ($O(n)$). Ще більше пам’яті на вказівники (`next` та `prev`).

Контрольні запитання

2. У чому полягає фундаментальна відмінність у зберіганні даних між масивом та зв’язним списком?
3. Поясніть, чому вставка на початок однозв’язного списку є операцією $O(1)$, а вставка в кінець — $O(n)$.
4. Яку ключову проблему однозв’язного списку вирішує двозв’язний список?
5. Наведіть приклад задачі, де циклічний зв’язний список буде значно зручнішим за звичайний.
6. Чому зв’язані списки вважаються менш кеш-ефективними, ніж масиви?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація двозв’язного списку та його основних операцій.

2. Мета

Набути практичних навичок у реалізації вузлових структур даних, що використовують вказівники. Зрозуміти механіку маніпуляції вказівниками для операцій вставки та видалення у двозв’язному списку.

3. Завдання

Створити клас `DoublyLinkedList`, який реалізує двозв’язний список для зберігання цілих чисел. Клас повинен надавати методи для додавання

елементів на початок і в кінець, видалення елемента за значенням, та виведення списку в прямому та зворотному порядку.

4. Опис

Клас `DoublyLinkedList` повинен мати два приватні поля-вказівники: `head` та `tail`. Внутрішня структура `DNode` повинна містити поле для даних та два вказівники: `next` та `prev`.

5. Порядок виконання

- Створіть структуру `DNode`: Опишіть вузол з полями `data`, `next`, `prev`.
- Створіть клас `DoublyLinkedList`: Додайте поля `head` та `tail`, ініціалізуйте їх як `NULL` в конструкторі.
- Реалізуйте `insertAtHead(data)`: Метод, що додає новий вузол на початок списку. Не забудьте коректно оновити вказівники `head`, `prev` та `next`.
- Реалізуйте `insertAtTail(data)`: Метод, що додає новий вузол в кінець списку. Ця операція має бути $O(1)$ завдяки наявності вказівника `tail`.
- Реалізуйте `deleteNode(value)`: Метод, що знаходить перший вузол із заданим значенням і видаляє його, коректно “зшиваючи” сусідні вузли.
- Реалізуйте обходи: Напишіть два методи: `printForward()` (рухається від `head` до `tail` по `next`) та `printBackward()` (рухається від `tail` до `head` по `prev`).
- Протестуйте: Створіть у функції `main` об’єкт вашого класу, додайте декілька елементів, видаліть один із середини, виведіть список в обох напрямках.

6. Результат

Повністю функціональний клас, що реалізує двозв’язний список з усіма основними операціями.

7. Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми з реалізацією класів `DNode` та `DoublyLinkedList`.
- Продемонструвати роботу додатку: показати вставку, видалення та виведення списку в прямому та зворотному порядку.

8. Бути готовим відповісти на питання:

- Пояснити на прикладі вашого коду, як відбувається зміна вказівників при видаленні вузла з середини списку.
- Чому ваша реалізація **`insertAtTail`** має складність $O(1)$?
- Які переваги та недоліки вашої реалізації порівняно з реалізацією списку на основі масиву?

Тести

1. Яка часова складність доступу до елемента за індексом **`k`** в однозв'язному списку з N елементів?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(k)$

2. Яка головна перевага двозв'язного списку над однозв'язним?

oo) Використовує менше пам'яті.

pp) Швидший довільний доступ.

qq) Ефективне видалення довільного вузла (при наявності вказівника на нього).

rr) Простіша реалізація.

3. У вас є вказівник на вузол **`p`** в середині однозв'язного списку. Яка складність операції видалення цього вузла?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) Неможливо виконати без додаткової інформації.

4. В якій структурі даних вказівник **`next`** останнього елемента вказує на перший елемент?

a) Стек

b) Двозв'язний список

c) Масив

d) Циклічний зв'язний список

5. Для якої операції зв’язаний список є значно ефективнішим за динамічний масив?

a) Отримання кількості елементів.

b) Пошук елемента за значенням.

c) Доступ до елемента за індексом.

d) Вставка елемента на початок послідовності.

Відповіді: 1-c (і d, оскільки k може бути близьким до N), 2-c, 3-c (бо потрібно знайти попередній вузол), 4-d, 5-d

Тести підвищеної складності

1. При вставці нового вузла **newNode** після вузла **node** у двозв’язному списку, який порядок операцій з вказівниками є єдиним коректним, щоб уникнути втрати частини списку? (Припускаємо, що **node** не є останнім елементом)

a) `node.next = newNode; newNode.prev = node; newNode.next = node.next.prev; node.next.prev = newNode;`

b) `newNode.next = node.next; newNode.prev = node; node.next.prev = newNode; node.next = newNode;`

c) `node.next = newNode; newNode.next = node.next.next; node.next.prev = newNode; newNode.prev = node;`

d) `newNode.prev = node; newNode.next = node.next; node.next = newNode; node.prev.next = newNode;`

2. Яка найбільш ефективна часова складність для знаходження середнього елемента в однозв’язному списку з N елементів, і який алгоритм для цього використовується?

a) $O(\log N)$, використовуючи варіацію двійкового пошуку.

b) $O(N)$, використовуючи метод двох вказівників (“черепаха та заєць”), які рухаються з різною швидкістю.

c) $O(N)$, шляхом першого проходу для підрахунку кількості елементів, а потім другого проходу до N/2.

d) $O(1)$, якщо зберігати окремий вказівник на середній елемент та оновлювати його при кожній вставці/видаленні.

3. Чому ітерація по великому масиву (1 мільйон елементів) на практиці зазвичай значно швидша, ніж ітерація по великому зв’язному списку

з такою ж кількістю елементів, хоча асимптотична складність обох операцій — $O(N)$?

a) Тому що операція розіменування вказівника (`->next`) є значно повільнішою на рівні процесора, ніж інкремент індексу масиву.

b) Тому що зв'язані списки завжди займають вдвічі більше пам'яті через вказівники, що сповільнює її читання.

c) Через кращу просторову локальність даних у масиві, що дозволяє кеш-пам'яті процесора ефективно завантажувати наперед сусідні елементи і мінімізувати дорогі звернення до оперативної пам'яті.

d) Тому що компілятор не може оптимізувати цикл для зв'язного списку.

4. Ви реалізуєте функцію історії переглянутих сторінок у веб-браузері, яка має підтримувати швидкі операції "Назад" та "Вперед". Яка структура даних буде найбільш ефективною для цього завдання?

a) Однозв'язний список, оскільки додавання нової сторінки відбувається швидко.

b) Циклічний зв'язний список, оскільки він дозволяє переходити з останньої сторінки на першу.

c) Двозв'язний список, оскільки він дозволяє переміщатися в обох напрямках (до `prev` для "Назад", до `next` для "Вперед") за час $O(1)$.

d) Динамічний масив, оскільки він надає швидкий доступ до будь-якої сторінки за її порядковим номером в історії.

5. Вам дано вказівник `p` на вузол, який потрібно видалити з однозв'язного списку. Ви не маєте доступу до голови списку, але гарантовано знаєте, що `p` не є останнім вузлом. Як можна видалити цей вузол зі складністю $O(1)$?

a) Це неможливо зробити, оскільки для видалення потрібен доступ до попереднього вузла, `p->prev`.

b) Скопіювати дані з наступного вузла (`p->next`) у поточний вузол `p`, а потім видалити наступний вузол.

c) Пройти по вказівниках `next` від вузла `p` до кінця списку, щоб знайти його хвіст, і потім працювати з ним.

d) Створити новий список без цього елемента, що є операцією $O(N)$.

Відповіді

1. b) Це єдиний коректний порядок. Спочатку потрібно “вплести” `newNode` між `node` та `node.next` (`newNode.next = node.next; newNode.prev = node;`), потім оновити вказівник `prev` у колишнього сусіда (`node.next.prev = newNode;`), і лише в кінці змінити `next` самого `node`. Будь-який інший порядок, наприклад, `node.next = newNode;` на першому кроці, призведе до втрати посилання на решту списку.

2. b) Класичний алгоритм “черепаха та заєць” (один вказівник рухається на 1 крок, інший — на 2) дозволяє знайти середину за один прохід, тобто за лінійний час $O(N)$. Варіант (c) також має складність $O(N)$, але вимагає двох проходів. Варіант (d) є теоретично можливим, але робить операції вставки/видалення дуже дорогими ($O(N)$), що нівелює переваги списку.

3. c) Елементи масиву лежать у пам’яті послідовно. Коли процесор зчитує один елемент, він автоматично завантажує в кеш і його сусідів (кеш-лінію). При ітерації наступні потрібні дані вже знаходяться у швидкому кеші. Вузли зв’язного списку розкидані по пам’яті, тому кожен перехід по вказівнику `next` з високою ймовірністю призводить до кеш-промаху та повільного звернення до RAM.

4. c) Двозв’язний список є ідеальною структурою для цього завдання. Кожен вузол (сторінка) має посилання на попередню та наступну сторінку, що робить переходи “Назад” (`prev`) та “Вперед” (`next`) миттєвими операціями $O(1)$.

5. b) Це відомий алгоритмічний трюк. Ми не можемо видалити сам вузол `p`, оскільки не маємо доступу до його попередника. Але ми можемо “перетворити” `p` на його наступника, скопіювавши дані, а потім видалити наступника, зв’язки до якого у нас є. Це ефективно видаляє значення вузла `p` зі списку.

Завдання для самостійної роботи

1. Реверс списку. Напишіть функцію, що “перевертає” однозв’язний список, не створюючи нового. Вона має змінити напрямок вказівників.
2. Пошук циклу. Реалізуйте алгоритм Флойда (“черепаха та заєць”) для визначення, чи містить зв’язний список цикл.
3. Середина списку. Напишіть функцію, що знаходить середній елемент однозв’язного списку за один прохід.

4. Стек на списку. Реалізуйте АТД “Стек”, використовуючи однозв’язний список як основу.
5. Черга на списку. Реалізуйте АТД “Черга”, використовуючи однозв’язний список з вказівниками на `head` та `tail`.
6. Злиття двох відсортованих списків. Напишіть функцію, що приймає два відсортовані зв’язні списки і повертає один новий відсортований список, що є їх злиттям.
7. Видалення дублікатів. Напишіть функцію, що видаляє дублікати з відсортованого зв’язного списку.
8. Паліндром. Напишіть функцію, що перевіряє, чи є однозв’язний список паліндромом.
9. Реалізація Циклічного списку. Реалізуйте клас для циклічного двозв’язного списку та функцію для його “розрізання” на два менших циклічних списки.
10. Видалення вузла без доступу до `head`. Вам дано вказівник лише на вузол `p` в середині однозв’язного списку (не на останній). Реалізуйте його видалення. (Підказка: це трюк).
11. “Розплющення” списку. Уявіть двозв’язний список, де кожен вузол, окрім `next` та `prev`, може мати вказівник `child` на інший двозв’язний список. Напишіть функцію, що “розплющує” таку структуру в один звичайний двозв’язний список.
12. Обертання списку. Напишіть функцію, що циклічно зсуває зв’язний список на *k* позицій.
13. Додавання чисел. Два числа представлені у вигляді зв’язаних списків, де кожен вузол — це одна цифра. Напишіть функцію, що додає ці два числа і повертає результат у вигляді нового зв’язного списку.

Стек

Ми вже розглянули фундаментальні способи зберігання послідовностей — масиви та зв'язані списки. Тепер ми перейдемо до вивчення абстрактних типів даних, які будуються на основі цих структур, але вводять строгі правила доступу до елементів. Перший і найважливіший з них — стек.

Теоретичні відомості

Ідея Стеку: Принцип LIFO (Last-In, First-Out)

Уявіть собі стопку книг. Ви можете легко покласти нову книгу зверху або взяти верхню книгу. Але якщо ви захочете дістати книгу з середини стопки, вам доведеться спершу зняти всі книги, що лежать над нею. Ця проста, але строга модель поведінки і є суттю стеку.

Стек — це абстрактний тип даних, що представляє собою лінійну колекцію елементів, яка працює за принципом LIFO (Last-In, First-Out), що означає "останнім прийшов, першим пішов". Елемент, який був доданий до стеку останнім, буде видалений з нього першим.

Кінець стеку, з якого відбуваються всі операції, називається його вершиною (top).

АТД "Стек": Контракт Як абстрактний тип даних, стек визначає не свою внутрішню будову, а "контракт" — набір операцій, які можна з ним виконувати:

- `push(item)`: Додати новий елемент на вершину стеку.
- `pop()`: Видалити елемент з вершини стеку та повернути його значення.
- `peek()` (або `top()`): Подивитись на елемент на вершині, не видаляючи його.
- `isEmpty()`: Перевірити, чи стек порожній.

Будь-яка структура, що реалізує ці чотири операції з дотриманням принципу LIFO, може називатися стеком.

Реалізація Стеку на основі масиву

Це один з найпоширеніших способів реалізації стеку.

Ідея: Ми використовуємо динамічний масив для зберігання елементів. Окрім самого масиву, нам потрібна лише одна додаткова змінна, яку ми назвемо `top_index`, що буде зберігати індекс останнього доданого елемента. Спочатку, коли стек порожній, `top_index` можна встановити в -1.

Як це працює:

- push(item): Щоб додати елемент, ми збільшуємо `top_index` на одиницю і записуємо новий елемент у масив за цим новим індексом. Якщо внутрішній масив заповнений, спершу відбувається його розширення (перерозподіл пам'яті).

– pop(): Щоб видалити елемент, ми повертаємо значення, що знаходиться за індексом **top_index**, а потім зменшуємо **top_index** на одиницю. Сам елемент фізично залишається в пам'яті масиву, але стає недоступним.

– peek(): Просто повертаємо елемент за індексом **top_index**.

– isEmpty(): Перевіряємо, чи дорівнює **top_index** -1.

Псевдокод реалізації на масиві:

```
КЛАС ArrayStack:
    ПРИВАТНІ ПОЛЯ:
        data      // Вказівник на внутрішній масив
        top_index // Індекс вершини стеку
        capacity  // Місткість масиву

    // Операція додавання
    ФУНКЦІЯ push(item):
        // (При необхідності тут відбувається логіка розширення масив
у,
        //  якщо top_index + 1 == capacity)
        top_index = top_index + 1
        data[top_index] = item
    КІНЕЦЬ ФУНКЦІЇ

    // Операція видалення
    ФУНКЦІЯ pop():
        ЯКЩО isEmpty() ТО ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Стек порожній")

        item_to_return = data[top_index]
        top_index = top_index - 1
        ПОВЕРНУТИ item_to_return
    КІНЕЦЬ ФУНКЦІЇ

    // Операція перегляду вершини
    ФУНКЦІЯ peek():
        ЯКЩО isEmpty() ТО ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Стек порожній")
        ПОВЕРНУТИ data[top_index]
    КІНЕЦЬ ФУНКЦІЇ

    // Перевірка на пустоту
    ФУНКЦІЯ isEmpty():
        ПОВЕРНУТИ (top_index == -1)
    КІНЕЦЬ ФУНКЦІЇ
КІНЕЦЬ КЛАСУ
```

Аналіз складності: Усі операції — `push`, `pop`, `peek`, `isEmpty` — виконуються за константний час, $O(1)$.

Реалізація Стеку на основі зв'язного списку

Альтернативний підхід полягає у використанні однозв'язного списку.

Ідея: Ми будемо вважати голову (`head`) нашого зв'язного списку вершиною стеку.

Як це працює:

- push(item): Додавання нового елемента на вершину стеку — це еквівалент операції вставки на початок зв'язного списку (`insertAtHead`). Ми створюємо новий вузол і робимо його новою головою списку.
- pop(): Видалення елемента з вершини — це еквівалент операції видалення з початку зв'язного списку (`deleteFromHead`). Ми робимо головою списку другий елемент, а старий видаляємо.
- peek(): Повертаємо дані, що зберігаються у вузлі `head`.
- isEmpty(): Перевіряємо, чи `head` дорівнює `NULL`.

Псевдокод реалізації на зв'язному списку:

```
КЛАС LinkedStack:
    ПРИВАТНІ ПОЛЯ:
        head // Вказівник на вузол, що є вершиною стеку

    // Операція додавання
    ФУНКЦІЯ push(item):
        // Створюємо новий вузол
        newNode = Створити_Вузол(item)
        // Новий вузол тепер вказує на стару вершину
        newNode.next = head
        // Новий вузол стає новою вершиною (головою)
        head = newNode
    КІНЕЦЬ ФУНКЦІЇ

    // Операція видалення
    ФУНКЦІЯ pop():
        ЯКЩО isEmpty() ТО ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Стек порожній")

        item_to_return = head.data
        node_to_delete = head
        // Вершиною стає наступний елемент
        head = head.next
```

```
        ВИДАЛИТИ node_to_delete
        ПОВЕРНУТИ item_to_return
    КІНЕЦЬ ФУНКЦІЇ

    // Операція перегляду вершини
    ФУНКЦІЯ peek():
        ЯКЩО isEmpty() ТО ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Стек порожній")
        ПОВЕРНУТИ head.data
    КІНЕЦЬ ФУНКЦІЇ

    // Перевірка на пустоту
    ФУНКЦІЯ isEmpty():
        ПОВЕРНУТИ (head == NULL)
    КІНЕЦЬ КЛАСУ
```

Аналіз складності: Усі операції `push`, `pop`, `peek`, `isEmpty` мають гарантовану складність $O(1)$, оскільки вони зачіпають лише перший елемент списку.

Порівняння реалізацій

Який же підхід кращий? Це класичний інженерний компроміс.

– Реалізація на масиві:
  - Переваги: Краща кеш-ефективність завдяки суцільному зберіганню даних. Менші витрати пам'яті на елемент (немає вказівників).
  - Недоліки: Можливі рідкісні, але дорогі операції перерозподілу пам'яті при розширенні. Може марнувати пам'ять, якщо `capacity` значно більше за `size`.

– Реалізація на зв'язному списку:
  - Переваги: Гарантований час $O(1)$ для всіх операцій. Пам'ять виділяється тільки тоді, коли це потрібно.
  - Недоліки: Гірша кеш-ефективність через розкиданість вузлів у пам'яті. Додаткові витрати пам'яті на зберігання вказівника `next` для кожного елемента.

На практиці для більшості завдань реалізація на основі динамічного масиву є стандартним вибором через кращу роботу з кешем процесора.

Область застосування

Обмеженість стеку є його головною силою. Він ідеально підходить для моделювання процесів, що мають вкладену або рекурсивну природу.

– Управління пам’яттю: Стек викликів для відстеження активних функцій, їхніх локальних змінних та адрес повернення.
– Синтаксичний аналіз: Перевірка правильності дужкових послідовностей, обчислення математичних виразів у зворотній польській нотації.
– Алгоритми: Пошук в глибину на графах або деревах може бути реалізований ітеративно за допомогою стеку.
– Історія дій: Реалізація функцій “Скасувати” у програмах.

Приклади застосування

– Кнопка “Назад” у браузері: Кожна відвідана сторінка додається у стек. Коли ви натискаєте “Назад”, остання сторінка видаляється, і ви повертаєтесь до попередньої.
– Обчислення виразу `(5 + 3) * 2`: Алгоритм на основі стеку може перетворити цей “інфіксний” запис у “постфіксний” (`5 3 + 2 *`), а потім легко обчислити його.
– Рекурсія: Кожен рекурсивний виклик функції насправді є операцією `push` у системний стек викликів. Коли функція завершується, відбувається `pop`.

Переваги та недоліки

Переваги

– Простота: Інтерфейс стеку є дуже простим та інтуїтивно зрозумілим.
– Передбачуваність: Принцип LIFO є строгим, що робить поведінку програм, що використовують стек, легкою для аналізу.
– Ефективність: Всі ключові операції виконуються за час $O(1)$.

Недоліки

– Обмежений доступ: Неможливо отримати доступ до елементів у середині стеку, не зруйнувавши його. Якщо вам потрібен такий доступ, стек — не ваш вибір.

Контрольні запитання

1. Сформулюйте принцип LIFO своїми словами. Наведіть аналогію з реального життя.
2. Які чотири основні операції визначає АТД “Стек”?

3. Опишіть, як працює операція `push` в реалізації стеку на основі масиву, особливо у випадку, коли масив заповнений.
4. Чому операція `pop` в реалізації на основі однозв'язного списку має складність $O(1)$?
5. Порівняйте дві реалізації стеку (на масиві та на списку) з точки зору використання пам'яті та кеш-ефективності.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація АТД "Стек" на основі масиву та зв'язного списку.

2. Мета

Набути практичних навичок у реалізації абстрактного типу даних "Стек" двома різними способами. Зрозуміти різницю між логічним інтерфейсом (АТД) та його фізичною реалізацією (структурою даних).

3. Завдання

Створити програму, яка реалізує стек для зберігання цілих чисел двома способами: на основі динамічного масиву та на основі однозв'язного списку. Програма має надати користувачеві можливість перевірити правильність роботи дужкової послідовності.

4. Опис

Спочатку потрібно створити єдиний інтерфейс (або абстрактний клас) `IStack` з методами `push`, `pop`, `peek`, `isEmpty`. Потім створити два класи, `ArrayStack` та `LinkedStack`, що реалізують цей інтерфейс. Основна програма має використовувати стек для перевірки дужок у введеному рядку.

5. Порядок виконання
    1. Створіть інтерфейс: Опишіть інтерфейс `IStack`.
    2. Реалізуйте `ArrayStack`: Створіть клас, що реалізує `IStack` на основі динамічного масиву, як описано в теоретичній частині.
    3. Реалізуйте `LinkedStack`: Створіть клас, що реалізує `IStack` на основі однозв'язного списку.
    4. Реалізуйте перевірку дужок: Напишіть функцію `checkBrackets(text)`, яка приймає рядок і використовує стек для перевірки правильності розстановки дужок `()`, `[]`, `{}`.

- *Алгоритм:* Ідучи по рядку, якщо зустрічаємо відкриваючу дужку, кладемо її у стек. Якщо зустрічаємо закриваючу, дістаємо елемент зі стеку і перевіряємо, чи утворюють вони правильну пару. Якщо наприкінці стек порожній — послідовність коректна.

5. Створіть тест: Напишіть функцію `main`, яка дозволяє користувачеві ввести рядок і перевірити його за допомогою вашої функції `checkBrackets`.

6. Результат

Додаток, що демонструє використання стеку для розв'язання практичної задачі та дозволяє порівняти дві його реалізації.

7. Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми з інтерфейсом та двома реалізаціями стеку.
- Продемонструвати роботу додатку на кількох прикладах дужкових послідовностей (коректних та некоректних).

8. Бути готовим відповісти на питання:

- Яка головна ідея АТД і як вона відображена у вашому коді?
- Пояснити логіку роботи вашого алгоритму перевірки дужок.
- Якби вам потрібно було реалізувати стек для зберігання мільйонів елементів, але з жорсткими обмеженнями по пам'яті, яку б реалізацію ви обрали і чому?

Тести

1. Який принцип лежить в основі роботи стеку?

a) FIFO (First-In, First-Out)

b) LIFO (Last-In, First-Out)

c) Довільний доступ

d) Сортування за пріоритетом

2. Яка операція не є стандартною для АТД "Стек"?

a) `push(item)`

b) `pop()`

c) `peek()`

d) `get(index)`

3. Яка гарантована часова складність операції `push` у реалізації стеку на зв'язному списку?

a) $O(1)$

b) $O(\log n)$

c) $O(n)$

d) Амортизована $O(1)$

4. Для якої з наступних задач стек є найбільш природним інструментом?

a) Управління чергою на друк.

b) Моделювання зв'язків у соціальній мережі.

c) Реалізація функції "Undo" (Скасувати) в текстовому редакторі.

d) Пошук товару в інтернет-магазині за артикулом.

5. При реалізації стеку на масиві, операція `pop` зазвичай...

a) Видаляє елемент з масиву фізично, зсовуючи інші елементи.

b) Просто зменшує значення індексу вершини стеку.

c) Завжди вимагає перерозподілу пам'яті.

d) Має складність $O(n)$.

Відповіді: 1-b, 2-d, 3-a, 4-c, 5-b

Тести підвищеної складності

1. При обчисленні постфіксного виразу `8 2 5 * + 1 3 2 * + -` за допомогою стека, яке значення залишиться у стеку після виконання всіх операцій?

a) 7

b) 11

c) 1

d) Помилка у виразі

2. Розглянемо наступний псевдокод. Яке значення поверне виклик `F(4)`?

```
ФУНКЦІЯ F(n):
    ЯКЩО n <= 1 ТО
        ПОВЕРНУТИ 1
    КІНЕЦЬ ЯКЩО
    ПОВЕРНУТИ n + F(n - 1)
КІНЕЦЬ ФУНКЦІЇ
```

a) 4

b) 9

c) 10

d) 24

3. Ви реалізуєте стек на основі однозв'язного списку. Яка стратегія реалізації операцій `push` та `pop` є найбільш ефективною, щоб обидві мали часову складність $O(1)$?

a) `push` — додавання в кінець списку, `pop` — видалення з кінця списку.

b) `push` — додавання на початок списку, `pop` — видалення з кінця списку.

c) `push` — додавання на початок списку, `pop` — видалення з початку списку.

d) `push` — додавання в кінець списку, `pop` — видалення з початку списку.

4. Програма перевіряє збалансованість дужок `()`, `[]`, `{}` у рядку `{[()()]}` за допомогою стека. Яким буде вміст стека (вершина праворуч) безпосередньо після обробки другої закриваючої дужки `)`?

a) `[{, [`

b) `[{]`

c) `[{, [, (`

d) Стек буде порожнім.

5. Який найбільш ефективний спосіб реалізувати два стеки в одному масиві фіксованого розміру `N` так, щоб помилка переповнення виникала лише тоді, коли сумарна кількість елементів обох стеків перевищить `N`?

a) Розділити масив навпіл: `[0...N/2-1]` для першого стека, `[N/2...N-1]` для другого.

b) Перший стек росте від індексу `0` вправо, а другий — від індексу `N-1` вліво. Переповнення відбувається, коли їхні вершини зустрічаються.

c) Зберігати елементи двох стеків по черзі (в парних та непарних індексах).

d) Це неможливо зробити ефективно, потрібно використовувати два окремі масиви.

Відповіді

1. b) 11

Трасування:

`[8]` → `[8, 2]` → `[8, 2, 5]` → `*` → `[8, 10]` → `+` → `[18]` → `[18, 1]` → `[18, 1, 3]` → `[18, 1, 3, 2]` → `*` → `[18, 1, 6]` → `+` → `[18, 7]` → `-` → `[11]`.

2. c) 10

Трасування: `F(1) = 1`. `F(2) = 2 + F(1) = 3`. `F(3) = 3 + F(2) = 6`. `F(4) = 4 + F(3) = 10`.

3. c) Операції **push** та **pop** на стеку мають відбуватися з одного і того ж кінця. В однозв'язному списку операції з головою (вставка та видалення) мають складність $O(1)$. Операції з хвостом (особливо видалення) вимагають проходу по всьому списку, що дає $O(n)$.

4. a) `{, [`

Трасування: `{` → `[{]`. `[` → `[{, []`. `(` → `[{, [, ()]`. `)` (перша) → **pop** → `[{, []`. `(` → `[{, [, ()]`. `)` (друга) → **pop** → `[{, []`.

5. b) Ця класична задача вирішується шляхом росту одного стека з індексу 0 вправо, а іншого — з індексу N-1 вліво. Це гарантує, що пам'ять використовується максимально ефективно, і переповнення (коли **top1 + 1 == top2**) станеться лише тоді, коли сумарна кількість елементів заповнить весь масив.

Завдання для самостійної роботи

1. Реалізація черги через два стеки. Реалізуйте АТД "Черга", використовуючи лише два стеки як внутрішню структуру даних. Проаналізуйте амортизаційну складність операцій **enqueue** та **dequeue**.
2. Сортування стеку. Напишіть функцію, яка сортує стек за зростанням (найменший елемент на вершині). Дозволяється використовувати лише один додатковий тимчасовий стек.
3. Обчислення виразу у зворотній польській нотації. Напишіть програму, яка обчислює значення арифметичного виразу, записаного у постфіксній нотації (напр., "5 3 + 2 *").

4. Min Stack. Розробіть стек, який, окрім стандартних операцій, підтримує функцію `getMin()`, що повертає мінімальний елемент у стеку за час $O(1)$.
5. Історія браузера. Створіть спрощену модель історії браузера, використовуючи два стеки (один для історії “назад”, інший для “вперед”).
6. Вежа Ханоя. Розв’яжіть класичну головоломку “Вежі Ханоя” за допомогою рекурсії. Проаналізуйте, яку роль у цьому розв’язку неявно відіграє системний стек викликів.
7. Ітеративний обхід дерева. Реалізуйте прямий (Pre-order) обхід бінарного дерева, використовуючи стек замість рекурсії.
8. Спрощення шляхів. Напишіть функцію, що приймає шлях у файловій системі (напр., `/a/b/../c/./d`) і повертає його спрощений канонічний вигляд (`/a/c/d`).
9. Пастки дощу. Вам дано масив невід’ємних чисел, що представляють висоти стовпчиків. Порахуйте, скільки води може затриматися між ними після дощу.
10. Гістограма. Знайдіть площу найбільшого прямокутника в гістограмі, представленій масивом висот. Ця класична задача елегантно розв’язується за допомогою стеку.

Черга, кільцева черга.

Розглянувши стек з його принципом "останнім прийшов, першим пішов", ми переходимо до його ідеологічного антипода — черги, яка втілює принцип справедливості та порядку.

Теоретичні відомості

Ідея Черги: Принцип FIFO (First-In, First-Out)

Якщо стек можна було уявити як стопку тарілок, то найкраща аналогія для черги — це черга людей до каси в магазині. Перша людина, яка стала в чергу, буде обслужена першою. Нові люди стають в кінець черги. Цей простий та справедливий принцип називається FIFO (First-In, First-Out), що означає “першим прийшов, першим пішов”.

Черга — це абстрактний тип даних, що представляє собою лінійну колекцію елементів, яка працює за принципом FIFO. Операції додавання

відбуваються в одному кінці черги, який називається хвостом (tail або rear), а операції видалення — в іншому, що називається головою (head або front).

АТД "Черга": Контракт Як і стек, черга визначається своїм "контрактом" — набором операцій, що гарантують дотримання принципу FIFO:

- `enqueue(item)`: Додати новий елемент у хвіст черги.
- `dequeue()`: Видалити елемент з голови черги та повернути його значення.
- `front()` (або `peek()`): Подивитись на елемент у голові черги, не видаляючи його.
- `isEmpty()`: Перевірити, чи черга порожня.

Будь-яка структура, що реалізує ці операції, є чергою. Давайте розглянемо, як це можна зробити.

Наївна реалізація на основі масиву: проблема "повзучої черги"

Перша ідея, що спадає на думку, — використати для зберігання елементів динамічний масив. Нам знадобляться два індекси: `head` для позначення початку черги та `tail` для позначення кінця.

Як це працює (і в чому проблема):

15. Ініціалізація: `head = 0`, `tail = 0`.
16. `enqueue(item)`: Ми додаємо елемент в `array[tail]` і зсуваємо хвіст: `tail = tail + 1`.
17. `dequeue()`: Ми повертаємо елемент `array[head]` і зсуваємо голову: `head = head + 1`.

Здається, все добре, обидві операції мають складність $O(1)$. Але що станеться після кількох операцій?

Уявіть масив на 8 елементів.

- `enqueue(A), enqueue(B), enqueue(C)`: `head=0, tail=3`. Масив: `[A, B, C, _, _, _, _, _]`
- `dequeue()`: `head=1, tail=3`. Масив: `[_, B, C, _, _, _, _, _]`. Елемент A видалено, але комірка 0 тепер "мертва".
- `enqueue(D), ..., enqueue(H)`: `head=1, tail=8`. Масив: `[_, B, C, D, E, F, G, H]`

Тепер `tail` досяг кінця масиву. Ми не можемо додати нові елементи, хоча на початку масиву є вільне місце! Ця проблема називається "повзуча черга",

оскільки обидва індекси постійно “повзуть” вправо, залишаючи за собою невикористаний простір.

Елегантне рішення: Кільцева черга

Щоб розв’язати проблему “повзучої черги”, ми можемо “замкнути” масив у кільце. Уявіть, що за останньою коміркою масиву знову йде перша.

Ідея: Коли індекс `head` або `tail` досягає кінця масиву, він просто “перестрибує” на початок. Це дуже легко реалізувати за допомогою операції взяття за модулем (%).

Якщо масив має місткість `capacity`, то наступна позиція після індексу `i` обчислюється як `(i + 1) % capacity`. Наприклад, для масиву місткістю 8, після індексу 7 йде `(7 + 1) % 8 = 0`.

Проблема розрізнення повного та порожнього стану: Виникає нова проблема: як відрізнити порожню чергу від повної? В обох випадках індекс `head` може збігатися з індексом `tail`. Існує два популярних рішення:

1. Зберігати лічильник: Тримати окрему змінну `size`. Черга порожня, якщо `size == 0`, і повна, якщо `size == capacity`. Це найнадійніший спосіб.
2. Залишати одну комірку порожньою: Вважати чергу повною, коли кількість елементів досягає `capacity - 1`. Умова заповненості: `(tail + 1) % capacity == head`.

Псевдокод реалізації кільцевої черги (з лічильником `size`):

```
КЛАС CircularQueue:
    ПРИВАТНІ ПОЛЯ:
        data      // Внутрішній масив
        head      // Індекс голови
        tail      // Індекс наступної вільної комірки
        size      // Поточна кількість елементів
        capacity  // Місткість масиву

    // Операція додавання
    ФУНКЦІЯ enqueue(item):
        ЯКЩО size == capacity ТО
            ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Черга повна")
        КІНЕЦЬ ЯКЩО

        data[tail] = item
        tail = (tail + 1) % capacity // Перехід по кільцю
        size = size + 1
```

```
    КІНЕЦЬ ФУНКЦІЇ

    // Операція видалення
    ФУНКЦІЯ dequeue():
        ЯКЩО isEmpty() ТО
            ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Черга порожня")
        КІНЕЦЬ ЯКЩО

        item_to_return = data[head]
        head = (head + 1) % capacity // Перехід по кільцю
        size = size - 1
        ПОВЕРНУТИ item_to_return
    КІНЕЦЬ ФУНКЦІЇ

    // ... інші методи: front(), isEmpty() ...
КІНЕЦЬ КЛАСУ
```

Складність: $O(1)$ для всіх операцій (з урахуванням амортизації для динамічного розширення масиву, якщо воно реалізоване).

Реалізація на основі зв'язного списку

Цей підхід є природним для черги і не має проблем з "повзучістю" чи переповненням (доки є пам'ять).

Ідея: Щоб забезпечити $O(1)$ для операцій з обох кінців, нам потрібно зберігати вказівники і на голову (`head`), і на хвіст (`tail`) однозв'язного списку.

Як це працює:

- `enqueue(item)`: Додаємо новий вузол після поточного хвоста (`tail`) і оновлюємо вказівник `tail` на цей новий вузол.
- `dequeue()`: Видаляємо вузол, на який вказує голова (`head`), і зсуваємо `head` на наступний елемент.

Псевдокод реалізації на зв'язному списку:

```
КЛАС LinkedQueue:
    ПРИВАТНІ ПОЛЯ:
        head // Вказівник на голову (початок) черги
        tail // Вказівник на хвіст (кінець) черги

    // Операція додавання
    ФУНКЦІЯ enqueue(item):
        newNode = Створити_Вузол(item)

        ЯКЩО isEmpty() ТО
```

```
            // Якщо черга порожня, голова і хвіст - це той самий вузо
л
            head = newNode
            tail = newNode
        ІНАКШЕ
            // Додаємо новий вузол після хвоста
            tail.next = newNode
            // Новий вузол стає новим хвостом
            tail = newNode
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ФУНКЦІЇ

    // Операція видалення
    ФУНКЦІЯ dequeue():
        ЯКЩО isEmpty() ТО ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Черга порожня")

        item_to_return = head.data
        node_to_delete = head
        head = head.next

        // Якщо після видалення черга стала порожньою
        ЯКЩО head == NULL ТО
            tail = NULL
        КІНЕЦЬ ЯКЩО

        ВИДАЛИТИ node_to_delete
        ПОВЕРНУТИ item_to_return
    КІНЕЦЬ ФУНКЦІЇ
КІНЕЦЬ КЛАСУ
```

Складність: $O(1)$ для всіх ключових операцій.

Область застосування

Принцип FIFO є фундаментальним для багатьох процесів в обчислювальних системах.

- Операційні системи: Планувальники завдань для процесора, управління запитами до пристроїв вводу-виводу (диск, принтер).
- Комп’ютерні мережі: Буферизація мережевих пакетів у маршрутизаторах. Пакети, що прийшли раніше, обробляються раніше.
- Алгоритми на графах: Пошук в ширину (BFS) використовує чергу для зберігання вершин, які потрібно відвідати.

– Симуляції та моделювання: Імітація реальних систем з чергами (системи масового обслуговування, логістика).

Приклади застосування

– Принтер: Коли ви відправляєте декілька документів на друк, вони стають у чергу і друкуються в тому порядку, в якому були відправлені.

– Обробка запитів на сервері: Веб-сервер може ставити вхідні запити від користувачів у чергу, щоб обробляти їх послідовно і не втратити жодного.

– Ігри: В онлайн-іграх система пошуку матчу (matchmaking) ставить гравців у чергу, перш ніж сформувати команду.

Переваги та недоліки

Переваги

– Справедливість та порядок: Гарантує, що задачі або дані будуть оброблені в тому порядку, в якому вони надійшли.

– Простота: Концепція FIFO є дуже простою для розуміння.

– Ефективність: Всі ключові операції виконуються за час $O(1)$ (для правильних реалізацій).

Недоліки

– Обмежений доступ: Як і стек, черга не надає доступу до елементів у середині. Якщо вам потрібно шукати або видаляти елементи з довільних позицій, черга не підходить.

Контрольні запитання

1. Сформулюйте принцип FIFO своїми словами. Наведіть аналогію з реального життя, відмінну від наведених.
2. Опишіть проблему “повзучої черги” при наївній реалізації на масиві.
3. Як операція взяття за модулем (%) допомагає реалізувати кільцеву чергу?
4. Чому для ефективної реалізації черги на зв’язному списку потрібно зберігати вказівники і на голову, і на хвіст?
5. Порівняйте реалізацію кільцевої черги на масиві та черги на зв’язному списку. Які їхні переваги та недоліки одна відносно одної?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація кільцевої черги на основі масиву.

2. Мета

Набути практичних навичок у реалізації АТД “Черга” з використанням кільцевого буфера для ефективного управління пам’яттю та досягнення складності $O(1)$ для операцій.

3. Завдання

Створити клас `CircularQueue`, який реалізує чергу для зберігання цілих чисел на основі масиву фіксованої місткості. Програма має коректно обробляти ситуації “переповнення” та “пустої черги”.

4. Опис.

Клас `CircularQueue` повинен мати приватні поля: масив для даних, індекси `head` та `tail`, а також лічильник `size` та змінну `capacity`.

5. Порядок виконання

1. Створіть клас: Опишіть клас `CircularQueue` з необхідними полями.
2. Реалізуйте конструктор: Він має приймати на вхід місткість, виділяти пам’ять для масиву та ініціалізувати `head=0, tail=0, size=0`.
3. Реалізуйте `enqueue(item)`: Керуючись псевдокодом, реалізуйте логіку додавання елемента. Перед додаванням перевіряйте, чи черга не повна (`size == capacity`). Оновлюйте `tail` за формулою `(tail + 1) % capacity`.
4. Реалізуйте `dequeue()`: Реалізуйте логіку видалення. Перед видаленням перевіряйте, чи черга не порожня. Оновлюйте `head` за формулою `(head + 1) % capacity`.
5. Реалізуйте `front()` та `isEmpty()`.
6. Створіть тест: Напишіть функцію `main`, яка тестує вашу чергу. Продемонструйте, як `head` та `tail` “перестрибують” через кінець масиву. Перевірте поведінку при спробі додати елемент у повну чергу та видалити з порожньої.

6. Результат

Повністю функціональний клас, що реалізує кільцеву чергу.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією класу `CircularQueue`.

2. Продемонструвати роботу додатку: показати, як змінюються `head`, `tail` та `size` під час виконання операцій, особливо при “переході через нуль”.
3. Бути готовим відповісти на питання: .....
    - Пояснити, як у вашому коді працює операція взяття за модулем для реалізації кільцевого буфера.
    - Як ви розрізняєте повну та порожню чергу у вашій реалізації?
    - Чому цей підхід ефективніший за “наївну” реалізацію черги на масиві?

Тести

1. Який принцип лежить в основі роботи черги?

a) FIFO (First-In, First-Out)

b) LIFO (Last-In, First-Out)

c) Довільний доступ

d) Сортування за пріоритетом

2. В кільцевій черзі на масиві операція `enqueue` додає елемент на позицію, що визначається індексом...

a) `head`

b) `tail`

c) `size`

d) `capacity - 1`

3. Яка головна проблема “наївної” реалізації черги на масиві?

a) Повільна вставка.

b) Повільне видалення.

c) Поступове “сповзання” даних в кінець масиву і марнування місця на початку.

d) Неможливість зберігати більше 10 елементів.

4. Для ефективної ($O(1)$) реалізації черги на зв’язному списку необхідно зберігати вказівники на:

a) Лише на голову (`head`).

b) Лише на хвіст (`tail`).

c) І на голову, і на хвіст.

d) На кожен елемент списку.

5. Для якого з алгоритмів черга є основною структурою даних?

a) Пошук в глибину (DFS).

b) Бінарний пошук.

c) Пошук в ширину (BFS).

d) Сортування злиттям.

Відповіді: 1-a, 2-b, 3-c, 4-c, 5-c

Тести підвищеної складності

1. Яку фундаментальну проблему наївної реалізації черги на звичайному масиві (де **`front`** та **`rear`** рухаються тільки вправо) вирішує кільцева черга?

a) Проблему повільної операції **`enqueue`** ($O(n)$).

b) Проблему неефективного використання пам'яті, коли після кількох операцій **`dequeue`** на початку масиву залишається багато вільного місця, яке не можна використати.

c) Проблему неможливості зберігати більше 100 елементів.

d) Проблему повільного доступу до елемента за індексом ($O(n)$).

2. Розглянемо кільцеву чергу розміром 5 (індекси 0-4). Початково **`front = 0`**, **`rear = 0`**. Виконується послідовність операцій: **`enqueue(10)`**, **`enqueue(20)`**, **`enqueue(30)`**, **`dequeue()`**, **`enqueue(40)`**. Якими будуть фінальні значення **`front`** та **`rear`**?

a) **`front = 0`**, **`rear = 4`**

b) **`front = 1`**, **`rear = 4`**

c) **`front = 1`**, **`rear = 3`**

d) **`front = 2`**, **`rear = 4`**

3. При реалізації кільцевої черги на масиві розміром **`N`**, коли вказівники **`front`** та **`rear`** збігаються (**`front == rear`**), виникає неоднозначність. Як її коректно вирішити, щоб відрізнити повну чергу від порожньої?

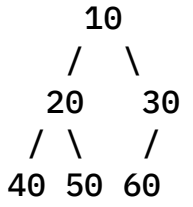
a) Завжди залишати одне місце в масиві вільним; черга вважається повною, коли **`(rear + 1) % N == front`**.

b) Використовувати додаткову змінну-лічильник **`size`**, яка зберігає поточну кількість елементів.

c) Використовувати додатковий булевий прапорець **`is_full`**.

d) Всі перелічені вище методи є коректними та широко вживаними.

4. При обході бінарного дерева в ширину (Level-Order Traversal) за допомогою черги, для дерева, зображеного нижче, що буде знаходитись у черзі одразу після того, як з неї буде вилучено вузол з ключем **`20`**?



a) **`[30, 40, 50]`**

b) **`[40, 50, 60]`**

c) **`[30, 40, 50, 60]`**

d) **`[30]`**

5. Яка структура даних є найбільш ефективною для реалізації дека (Deque), двосторонньої черги, що дозволяє додавати та видаляти елементи з обох кінців за час $O(1)$?

a) Однозв'язний список із вказівником на голову.

b) Динамічний масив (вектор).

c) Двозв'язний список із вказівниками на голову та хвіст.

d) Статичний масив.

Відповіді

1. b) У наївній реалізації вказівник **`rear`** рухається до кінця масиву, і навіть якщо **`front`** також зсувається (звільняючи місце на початку), нові елементи не можуть бути додані в цей звільнений простір. Кільцева черга вирішує це, "замикаючи" кінець масиву з його початком за допомогою операції взяття за модулем.

2. b) **`front = 1`**, **`rear = 4`**

*Трасування (припускаємо, **`rear`** вказує на наступну вільну комірку):*

1. **`enqueue(10)`**: **`A[0]=10`**, **`rear=1`**. Стан: **`f=0, r=1`**.
2. **`enqueue(20)`**: **`A[1]=20`**, **`rear=2`**. Стан: **`f=0, r=2`**.

3. `enqueue(30)`: `A[2]=30`, `rear=3`. Стан: `f=0, r=3`.
4. `dequeue()`: вилучаємо `A[0]`, `front=1`. Стан: `f=1, r=3`.
5. `enqueue(40)`: `A[3]=40`, `rear=4`. Стан: `f=1, r=4`.

3. d) Всі три підходи є канонічними способами розв'язання цієї проблеми. Використання лічильника `size` є найбільш прямолінійним, тоді як залишення одного вільного слота є елегантним трюком, що дозволяє обійтися лише вказівниками `front` та `rear`.

4. a) `[30, 40, 50]`

Трасування:

6. Початковий стан: `enqueue(10)`. Черга: `[10]`.
7. `dequeue(10)`, `enqueue` його нащадків (20, 30). Черга: `[20, 30]`.
8. `dequeue(20)`, `enqueue` його нащадків (40, 50). Черга: `[30, 40, 50]`. Це і є стан після вилучення вузла 20.

5. c) Двозв'язний список дозволяє миттєво ($O(1)$) отримувати доступ до обох кінців (через вказівники `head` та `tail`) і виконувати операції вставки/видалення, змінюючи лише сусідні вказівники. У динамічному масиві операції з початком масиву мають складність $O(n)$.

Завдання для самостійної роботи

1. Реалізація Стеку через дві черги. Реалізуйте АТД "Стек", використовуючи лише дві черги як внутрішню структуру даних.
2. Реалізація черги через два стеки. Реалізуйте АТД "Черга", використовуючи лише два стеки. Проаналізуйте амортизаційну складність операцій.
3. Реверс черги. Напишіть функцію, яка перевертає порядок елементів у черзі. Дозволяється використовувати стек.
4. Дек (Двостороння черга). Реалізуйте АТД "Дек" (`Deque`), що дозволяє додавати та видаляти елементи з обох кінців за $O(1)$. Використайте двозв'язний список як основу.
5. Генерація двійкових чисел. Напишіть програму, яка генерує та виводить двійкові числа від 1 до N, використовуючи чергу.
6. "Гаряча картоплина". Реалізуйте дитячу гру "Гаряча картоплина". N дітей стають у коло. Картоплина передається M разів. Дитина, у якої картоплина опинилася на M-й передачі, вибуває. Процес

повторюється, доки не залишиться один переможець. Використайте кільцеву чергу.

7. Рівневий обхід дерева. Напишіть функцію, що виконує обхід бінарного дерева за рівнями (level-order traversal), використовуючи чергу.
8. Черга з пріоритетом. Дослідіть структуру даних “Черга з пріоритетом” (Priority Queue). Чим вона відрізняється від звичайної черги? На якій структурі даних її зазвичай реалізують?
9. Ковзне вікно. Вам дано масив чисел та розмір вікна k. Для кожного вікна, що зсувається по масиву, знайдіть максимум. Розв’яжіть цю задачу ефективно, використовуючи дек.
10. Перший унікальний символ у потоці. Уявіть, що ви отримуєте потік символів. Розробіть алгоритм, який у будь-який момент може ефективно знайти перший символ, що ще не повторювався. Використайте чергу та хеш-таблицю.
11. Реалізація `Josephus Problem`. Дослідіть та реалізуйте розв’язок “Задачі Йосифа Флавія” за допомогою кільцевої черги.
12. Черга з фіксованою місткістю. Модифікуйте вашу реалізацію `LinkedQueue` так, щоб вона мала максимальну місткість і не дозволяла додавати більше елементів, ніж задано.
13. Порівняння реалізацій. Проведіть емпіричний тест, який порівнює швидкість мільйона операцій `enqueue/dequeue` для реалізації на масиві та на зв’язному списку. Поясніть результати.
14. Потокобезпечна черга. Дослідіть, які проблеми виникають при використанні черги в багатопотоковому середовищі та які механізми (м’ютекси, семафори) потрібні для створення потокобезпечної (thread-safe) черги.
15. Есе-аналіз. Напишіть есе на тему “Черги в операційних системах: від планування процесів до обробки переривань”.

Дек

Ми вже розглянули стек, що працює за принципом LIFO, та чергу з її принципом FIFO. Тепер ми познайомимось з їхнім потужним та гнучким “родичем” — деком, який поєднує в собі можливості обох цих структур.

Теоретичні відомості

Ідея Деку: Гібрид Стеку та Черги

Уявіть собі залізничну колію, що має виходи з обох боків. Вагони можуть приєднуватися та від'єднуватися як спереду, так і ззаду. Ця аналогія чудово ілюструє суть деку (Deque), назва якого є скороченням від Double-Ended Queue, або двостороння черга.

Дек — це абстрактний тип даних, що узагальнює стек та чергу. Він представляє собою лінійну колекцію елементів, в якій додавання та видалення елементів можна здійснювати з обох кінців: з голови (front) та з хвоста (tail).

Ця гнучкість робить дек своєрідним "швейцарським ножем" серед лінійних структур.

- Якщо використовувати лише операції **`addFirst`** та **`removeFirst`**, дек поводитиметься як стек.
- Якщо використовувати **`addLast`** та **`removeFirst`**, він поводитиметься як черга.

АТД "Дек": Контракт Оскільки дек є більш загальною структурою, його інтерфейс містить операції для обох кінців:

- **`addFirst(item)`** (або **`push_front`**): Додати новий елемент на початок (в голову) деку.
- **`addLast(item)`** (або **`push_back`**): Додати новий елемент в кінець (у хвіст) деку.
- **`removeFirst()`** (або **`pop_front`**): Видалити та повернути елемент з початку деку.
- **`removeLast()`** (або **`pop_back`**): Видалити та повернути елемент з кінця деку.
- **`getFirst()`** (або **`front`**): Подивитись на перший елемент, не видаляючи його.
- **`getLast()`** (або **`back`**): Подивитись на останній елемент, не видаляючи його.
- **`isEmpty()`**: Перевірити, чи дек порожній.

Реалізація на основі двозв'язного списку

Це найбільш природний та інтуїтивно зрозумілий спосіб реалізації деку.

Ідея: Двозв'язний список, який ми вже розглядали, ідеально підходить для цієї задачі. Наявність вказівників **`next`** та **`prev`** у кожному вузлі, а також

вказівників `head` та `tail` на рівні всього списку, дозволяє виконувати всі операції з обох кінців за константний час.

Як це працює:

- `addFirst(item)` повністю еквівалентна операції `insertAtHead` для двозв'язного списку.
- `addLast(item)` — це `insertAtTail`.
- `removeFirst()` — це `deleteFromHead`.
- `removeLast()` — це `deleteFromTail`.

Псевдокод реалізації на двозв'язному списку:

```
КЛАС LinkedDeque:
    ПРИВАТНІ ПОЛЯ:
        head // Вказівник на голову
        tail // Вказівник на хвіст

    // Операція додавання на початок
    ФУНКЦІЯ addFirst(item):
        newNode = Створити_Вузол(item)
        newNode.next = head
        ЯКЩО head != NULL ТО
            head.prev = newNode
        КІНЕЦЬ ЯКЩО
        head = newNode
        ЯКЩО tail == NULL ТО tail = newNode
    КІНЕЦЬ ФУНКЦІЇ

    // Операція додавання в кінець
    ФУНКЦІЯ addLast(item):
        newNode = Створити_Вузол(item)
        newNode.prev = tail
        ЯКЩО tail != NULL ТО
            tail.next = newNode
        КІНЕЦЬ ЯКЩО
        tail = newNode
        ЯКЩО head == NULL ТО head = newNode
    КІНЕЦЬ ФУНКЦІЇ

    // Операція видалення з початку
    ФУНКЦІЯ removeFirst():
        ЯКЩО isEmpty() ТО ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Дек порожній")
        item_to_return = head.data
        node_to_delete = head
```

```
        head = head.next
        ЯКЩО head != NULL ТО
            head.prev = NULL
        ІНАКШЕ
            tail = NULL // Дек став порожнім
        КІНЕЦЬ ЯКЩО
        ВИДАЛИТИ node_to_delete
        ПОВЕРНУТИ item_to_return
    КІНЕЦЬ ФУНКЦІЇ

    // ... аналогічно для removeLast, getFirst, getLast, isEmpty ...
КІНЕЦЬ КЛАСУ
```

Аналіз складності: Всі шість основних операцій (**addFirst**, **addLast**, **removeFirst**, **removeLast**, **getFirst**, **getLast**) мають гарантовану складність $O(1)$.

Реалізація на основі кільцевого масиву

Цей підхід є складнішим, але може бути більш ефективним з точки зору використання кеш-пам'яті.

Ідея: Ми використовуємо ідею кільцевого буфера, яку ми розглядали для черги, але розширюємо її. Нам так само потрібні індекси **head** та **tail**, але тепер вони можуть рухатись в обидва боки.

Як це працює:

- **addLast(item)**: Працює так само, як **enqueue** в кільцевій черзі. Елемент додається на позицію **tail**, і **tail** зсувається вперед по кільцю: **tail = (tail + 1) % capacity**.
- **removeFirst()**: Працює так само, як **dequeue**. Елемент забирається з позиції **head**, і **head** зсувається вперед: **head = (head + 1) % capacity**.
- **addFirst(item)**: Це нова, цікава операція. Щоб додати елемент на початок, нам потрібно зсунути **head** назад. Це робиться за допомогою специфічної модульної арифметики, щоб уникнути від'ємних індексів.
- **removeLast()**: Аналогічно, щоб видалити елемент з кінця, нам потрібно зсунути **tail** назад.

Псевдокод для операцій додавання/видалення на початок:

```
// Клас ArrayDeque має поля data, head, tail, size, capacity
// ...
```

```
// Операція додавання на початок
ФУНКЦІЯ addFirst(item):
    ЯКЩО size == capacity ТО ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Дек повний")

    // Зсуваємо head на одну позицію назад по кільцю
    head = (head - 1 + capacity) % capacity
    // Пояснення: `+ capacity` потрібно, щоб уникнути від'ємного резу
льтату,
    // якщо head був 0. Наприклад, (0 - 1 + 8) % 8 = 7.

    data[head] = item
    size = size + 1
КІНЕЦЬ ФУНКЦІЇ

// Операція видалення з кінця
ФУНКЦІЯ removeLast():
    ЯКЩО isEmpty() ТО ПОВІДОМИТИ_ПРО_ПОМИЛКУ("Дек порожній")

    // Зсуваємо tail на одну позицію назад по кільцю
    tail = (tail - 1 + capacity) % capacity

    item_to_return = data[tail]
    size = size - 1
    ПОВЕРНУТИ item_to_return
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності: Як і у випадку з динамічним масивом, операції додавання мають амортизовану складність $O(1)$ через можливі перерозподіли пам’яті. Операції видалення мають складність $O(1)$.

Область застосування

Дек є надзвичайно гнучкою структурою даних, що знаходить застосування в різноманітних задачах.

- Алгоритми: Особливо в задачах на графах та масивах.
- Планувальники завдань: Реалізація “work-stealing” планувальників, де процесори можуть “красти” завдання один в одного як з початку, так і з кінця черги, щоб збалансувати навантаження.
- Структури даних: Дек є основою для реалізації інших, більш складних структур.
- Історія операцій: Зберігання історії змін для реалізації функцій “Undo/Redo”, де нові дії додаються в один кінець, а скасування

відбувається з того ж кінця, але іноді потрібно обрізати “майбутню” історію з іншого кінця.

Приклади застосування

- Алгоритм “Ковзне вікно”: Для задач типу “знайти максимум у кожному підмасиві розміром k” дек є ідеальним інструментом, що дозволяє досягти лінійної складності $O(n)$. У деку зберігаються індекси потенційних максимумів.
- Пошук в ширину 0-1: Модифікація алгоритму BFS на графах, де ребра мають вагу 0 або 1. Ребра з вагою 0 обробляються пріоритетніше шляхом додавання вершин на початок деку, а з вагою 1 — в кінець.

Переваги та недоліки

Переваги

- Універсальність: Поєднує в собі функціональність стеку та черги, є більш гнучким.
- Ефективність: Всі операції на обох кінцях виконуються за час $O(1)$ (або амортизований $O(1)$).

Недоліки

- Обмежений доступ: Як і стек та черга, не надає ефективного доступу до елементів у середині. Пошук елемента або доступ за індексом залишаються операціями складності $O(n)$.
- Складніша реалізація: Реалізація на кільцевому масиві є складнішою за реалізацію звичайної кільцевої черги через необхідність коректно обробляти рух `head` та `tail` в обидва боки.

Контрольні запитання

1. Дайте визначення АТД “Дек”. Які операції він підтримує?
2. Поясніть, як за допомогою деку можна симулювати роботу стеку та черги.
3. Чому реалізація деку на двозв’язному списку є більш “природною” та простою, ніж на масиві?
4. Поясніть формулу `head = (head - 1 + capacity) % capacity` для операції `addFirst` в реалізації на кільцевому масиві. Чому тут необхідно додавати `capacity`?
5. Наведіть приклад задачі, де можливість додавати та видаляти елементи з обох кінців є ключовою перевагою.

Методичні вказівки до лабораторної роботи

1 Назва

Реалізація АТД “Дек” на основі двозв’язного списку.

2. Мета

Набути навичок у реалізації більш складних абстрактних типів даних. Продемонструвати, як одна структура даних (двозв’язний список) може слугувати ефективною основою для реалізації АТД (дек).

3. Завдання

Створити клас **`LinkedDeque`**, який реалізує дек для зберігання цілих чисел. Клас повинен надавати повний набір операцій: **`addFirst`**, **`addLast`**, **`removeFirst`**, **`removeLast`**, **`getFirst`**, **`getLast`**, **`isEmpty`**.

4. Опис.

Клас **`LinkedDeque`** має базуватися на реалізації двозв’язного списку з вказівниками на голову (**`head`**) та хвіст (**`tail`**). Всі операції деку мають бути реалізовані через відповідні операції над списком.

5. Порядок виконання

1. Створіть вузол: Опишіть структуру **`DNode`** з полями **`data`**, **`next`**, **`prev`**.
2. Створіть клас **`LinkedDeque`**: Додайте поля **`head`** та **`tail`**.
3. Реалізуйте операції додавання: Напишіть методи **`addFirst`** та **`addLast`**, керуючись псевдокодом. Зверніть особливу увагу на обробку випадку, коли дек є порожнім.
4. Реалізуйте операції видалення: Напишіть методи **`removeFirst`** та **`removeLast`**. Зверніть увагу на обробку випадку, коли в деку залишається один елемент і після видалення він стає порожнім.
5. Реалізуйте операції доступу: Напишіть методи **`getFirst`**, **`getLast`** та **`isEmpty`**.
6. Створіть тест: Напишіть функцію **`main`**, яка демонструє роботу всіх шести операцій. Наприклад: додайте елементи в обидва кінці, потім видаліть їх з обох кінців, перевіряючи значення на кожному кроці.

6. Результат

Повністю функціональний клас, що реалізує дек з операціями, які мають складність $O(1)$.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією класу `LinkedDeque`.
2. Продемонструвати роботу додатку: показати послідовне виконання всіх шести операцій, коментуючи стан деку після кожної.
3. Бути готовим відповісти на питання: .....
    - Пояснити, чому для реалізації деку був обраний саме двозв'язний, а не однозв'язний список.
    - Які переваги та недоліки вашої реалізації порівняно з гіпотетичною реалізацією на кільцевому масиві?
    - Як можна було б реалізувати стек, використовуючи лише ваш клас `LinkedDeque`?

Тести

1. Яка з наступних пар операцій НЕ може симулювати звичайну чергу (FIFO)?

a) `addLast`, `removeFirst`

b) `addFirst`, `removeLast`

c) `addLast`, `removeLast`

d) Всі пари a) та b) можуть симулювати чергу.

2. Яка структура даних є найбільш природною та ефективною для реалізації деку?

a) Статичний масив

b) Однозв'язний список

c) Двозв'язний список

d) Хеш-таблиця

3. Яка гарантована часова складність операції `addFirst` у реалізації деку на кільцевому масиві?

a) $O(1)$

b) O(log n)

c) $O(n)$

d) Амортизована $O(1)$

4. В реалізації деку на кільцевому масиві з місткістю N, якщо `head = 0`, куди вказуватиме `head` після операції `addFirst`?

a) -1

b) 1

c) N-1

d) N

5. Для якої задачі дек є класичним інструментом розв’язання?

a) Сортування масиву.

b) Пошук елемента в словнику.

c) Задача про ковзний максимум/мінімум.

d) Множення матриць.

Відповіді: 1-c, 2-c, 3-d, 4-c, 5-c

Тести підвищеної складності

1. Яка з наведених структур даних є найбільш ефективною для реалізації дека, що гарантує часову складність $O(1)$ для всіх чотирьох операцій (додавання/видалення з обох кінців)?

a) Статичний масив, де голова знаходиться на індексі 0, а хвіст — в кінці.

b) Однозв’язний список із вказівниками на голову та хвіст.

c) Двозв’язний список із вказівниками на голову та хвіст.

d) Динамічний масив (вектор).

2. Розглянемо дек, реалізований на кільцевому масиві розміром 5 (індекси 0-4). Початково він порожній. Виконується послідовність операцій: `addLast(10)`, `addLast(20)`, `removeFirst()`, `addLast(30)`, `addFirst(40)`. Яким буде фінальний вміст дека, якщо читати його з початку до кінця?

a) `[10, 20, 30, 40]`

b) `[40, 20, 30]`

c) `[30, 20, 40]`

d) `[40, 10, 20, 30]`

5. Для розв’язання задачі “пошук максимуму у ковзному вікні” використовується дек. Для масиву `[1, 3, -1, -3, 5, 3]` та вікна розміром 3, яким буде вміст дека (зберігаються індекси) одразу після обробки елемента `5` (індекс 4)?

a) `[4]`

b) `[1, 4]`

c) `[1, 2, 3]`

d) `[3, 4]`

4. Яку з наступних послідовностей операцій над деком неможливо відтворити, використовуючи лише один стек або одну чергу?

a) `addLast(10)`, `addLast(20)`, `removeLast()`, `removeLast()`

b) `addFirst(10)`, `addFirst(20)`, `removeFirst()`, `removeFirst()`

c) `addLast(10)`, `addLast(20)`, `removeFirst()`, `removeFirst()`

d) `addLast(10)`, `addFirst(20)`, `removeLast()`, `removeFirst()`

5. Використовуючи дек, реалізовано алгоритм перевірки рядка на паліндром. Який з наступних псевдокодів є коректним та ефективним?

a) Додати всі символи в дек. Потім у циклі одночасно видаляти по два елементи (`removeFirst` та `removeLast`) і порівнювати їх.

b) Додати всі символи в дек. Потім вилучити всі елементи через `removeFirst` в один рядок, а потім через `removeLast` в інший, і порівняти рядки.

c) Додати всі символи в дек. Потім, поки в деку більше одного елемента, робити `removeFirst` та `removeLast` і порівнювати їх. Якщо вони нерівні, рядок не є паліндромом.

d) Розділити рядок навпіл; першу половину додати в дек через `addLast`, а другу — через `addFirst`. Потім перевірити, чи дек порожній.

Відповіді

1) c Двозв'язний список є ідеальною основою, оскільки вказівники `head` та `tail` надають доступ до обох кінців за $O(1)$, а вказівники `next` та `prev` дозволяють виконати вставку/видалення також за $O(1)$. У динамічному масиві операції з початком мають складність $O(n)$.

`2) b [40, 20, 30]` *Трасування (конвенція: **`front`** — індекс першого, **`rear`** — індекс наступної вільної комірки):*

1. `init`: `f=0, r=0, size=0`.
2. `addLast(10)`: `A[0]=10`. `r=1, size=1`.
3. `addLast(20)`: `A[1]=20`. `r=2, size=2`.
4. `removeFirst()`: вилучаємо `A[0]`. `f=1, size=1`.
5. `addLast(30)`: `A[2]=30`. `r=3, size=2`.

6. `addFirst(40)`: `f=(1-1+5)%5=0`. `A[0]=40`. `f=0, size=3`. *Фінальний стан:* `front=0`, `rear=3`, `size=3`. Елементи: `A[0]=40`, `A[1]=20`, `A[2]=30`.

3) a [4] *Трасування алгоритму (в деку зберігаються індекси, він підтримується відсортованим за значеннями):*

1. `1` (інд 0): `deque = [0]`
2. `3` (інд 1): `3 > A[0]`. Вилучаємо 0. `deque = [1]`
3. `-1` (інд 2): `-1 < A[1]`. Додаємо 2. `deque = [1, 2]`
4. `-3` (інд 3): `-3 < A[2]`. Додаємо 3. `deque = [1, 2, 3]`
5. `5` (інд 4): `5 > A[3]`. Вилучаємо 3. `5 > A[2]`. Вилучаємо 2. `5 > A[1]`. Вилучаємо 1. Дек стає порожнім. Додаємо 4. `deque = [4]`.

4) d Послідовність `addLast`, `addFirst`, `removeLast`, `removeFirst` перевіряє обидва кінці дека. `addLast(10)` → `[10]`. `addFirst(20)` → `[20, 10]`. `removeLast()` → `[20]` (вилучили 10). `removeFirst()` → `[]` (вилучили 20). Цю послідовність неможливо симулювати ні стеком (де можна працювати лише з одним кінцем), ні чергою (де додавання і видалення відбуваються з різних кінців). Варіант (a) і (b) - це поведінка стека, варіант (c) - це поведінка черги.

5) c Це найбільш прямий та ефективний спосіб. Ми заповнюємо дек один раз за $O(N)$, а потім у циклі, що виконується N/2 разів, робимо дві дешеві операції ($O(1)$) та одне порівняння. Загальна складність — $O(N)$. Варіант (b) вимагає створення двох додаткових рядків та зайвих ітерацій.

Завдання для самостійної роботи

1. Реалізація на масиві. Реалізуйте АТД “Дек” на основі кільцевого масиву. Не забудьте про механізм розширення масиву, коли він заповнюється.
2. Паліндром. Напишіть функцію, що перевіряє, чи є рядок паліндромом, використовуючи дек.
3. Підмножини. Напишіть функцію, що генерує всі підмножини заданого масиву за допомогою деку.
4. Стек на деку. Реалізуйте АТД “Стек”, використовуючи ваш клас `LinkedDeque` як внутрішню структуру.

5. Черга на деку. Реалізуйте АТД “Черга”, використовуючи ваш клас `LinkedDeque`.
6. Ковзний мінімум. Реалізуйте розв’язок задачі “Ковзний мінімум” (знайти мінімум у кожному підмасиві розміром k) за час $O(n)$.
7. Аналіз реалізацій. Порівняйте продуктивність вашої реалізації деку на зв’язному списку та на масиві, виконавши велику кількість операцій додавання/видалення з обох кінців.
8. Обмежений дек. Створіть клас `BoundedDeque`, який має фіксовану максимальну місткість і не дозволяє додавати більше елементів.
9. Переміщення елементів. Напишіть функцію `rotate(k)`, яка зсуває всі елементи деку на *k* позицій вправо. Елементи, що “виходять” з одного кінця, мають з’являтися на іншому. Спробуйте реалізувати це ефективно.
10. Есе-аналіз. Напишіть есе на тему “Дек як узагальнена лінійна структура: коли його гнучкість є перевагою, а коли — надмірністю?”.

Множина, мультимножина, мішок як абстрактні типи.

Ми вже розглянули лінійні структури, де порядок елементів є ключовим. Тепер ми перейдемо до іншого типу колекцій, де головну роль відіграє не позиція, а сам вміст — унікальність та належність. Цей розділ присвячений абстрактним типам даних “Множина”, “Мультимножина” та “Мішок”.

Теоретичні відомості

Від Порядку до Унікальності: Вступ до АТД “Множина”

Досі ми розглядали структури, де порядок елементів був фундаментальним. У масиві ми звертаємось до елемента за індексом, у стеку та черзі порядок додавання визначає порядок видалення. Але що, якби нам була потрібна колекція, де порядок не має жодного значення, а головною є сама присутність або відсутність елемента?

Уявіть список гостей, запрошених на вечірку. Вам неважливо, в якому порядку ви записували їхні імена. Вам важливо знати, *хто* запрошений, і ви точно не можете запросити одну й ту саму людину двічі. Ця ідея — невпорядкована колекція унікальних елементів — і є суттю множини (Set).

На відміну від списку, де головне питання — “що знаходиться на позиції *i*?”, для множини головне питання — “чи *міститься* елемент *X* у цій колекції?”.

АТД “Множина” (Set): Контракт та Операції

Множина — це абстрактний тип даних, що представляє собою колекцію унікальних елементів без визначеного порядку.

Контракт АТД “Множина”:

- Дані: Колекція елементів, де кожен елемент є унікальним.
- Основні операції:
  - **`add(item)`**: Додає елемент до множини. Якщо такий елемент вже існує, множина не змінюється.
  - **`remove(item)`**: Видаляє елемент з множини, якщо він там є.
  - **`contains(item)`**: Перевіряє, чи міститься елемент у множині. Повертає **`true`** або **`false`**.
  - **`size()`**: Повертає кількість унікальних елементів у множині.

Окрім базових, множина підтримує класичні теоретико-множинні операції, які є надзвичайно корисними на практиці.

- Об’єднання (Union): Створює нову множину, що містить усі елементи з двох початкових множин.

```
ФУНКЦІЯ Union(Set A, Set B):
    newSet = Створити_копію(A)
    ДЛЯ кожного element B B:
        newSet.add(element)
    КІНЕЦЬ ДЛЯ
    ПОВЕРНУТИ newSet
КІНЕЦЬ ФУНКЦІЇ
```

- Перетин (Intersection): Створює нову множину, що містить лише ті елементи, які є одночасно в обох множинах.

```
ФУНКЦІЯ Intersection(Set A, Set B):
    newSet = Створити_порожню_множину()
    // Для ефективності ітеруємо по меншій множині
    smallerSet, largerSet = (A, B) ЯКЩО A.size() < B.size() ІНАК
ШЕ (B, A)

    ДЛЯ кожного element B smallerSet:
        ЯКЩО largerSet.contains(element) ТО
            newSet.add(element)
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
    ПОВЕРНУТИ newSet
КІНЕЦЬ ФУНКЦІЇ
```

– Різниця (Difference): Створює нову множину, що містить елементи, які є в першій множині, але відсутні у другій.

Реалізації Множини: Швидкість проти Порядку

АТД “Множина” можна реалізувати за допомогою різних структур даних, і вибір конкретної реалізації залежить від вимог до продуктивності та функціональності.

Реалізація 1: `HashSet` (на основі хеш-таблиці)

– Ідея: Використовувати хеш-таблицю, де елементи, що додаються, є одночасно і ключами, і значеннями (або значення просто ігнорується).

– Як це працює: Коли ми додаємо елемент, ми обчислюємо його хеш-код, щоб миттєво знайти його місце в таблиці. Це дозволяє дуже швидко перевірити, чи елемент вже існує.

– Профіль продуктивності:

  • `add`, `remove`, `contains`: $O(1)$ в середньому. Це найшвидший варіант.

– Особливість: Елементи зберігаються у невпорядкованому вигляді, що залежить від хеш-функції. Ітерація по такій множині не гарантує жодного певного порядку.

Реалізація 2: `TreeSet` (на основі збалансованого дерева пошуку)

– Ідея: Використовувати збалансоване бінарне дерево пошуку (наприклад, червоно-чорне дерево) для зберігання елементів.

– Як це працює: Елементи додаються в дерево відповідно до їхнього значення, підтримуючи властивості дерева пошуку. Унікальність гарантується самою структурою BST.

– Профіль продуктивності:

  • `add`, `remove`, `contains`: $O(\log n)$. Це повільніше, ніж у `HashSet`, але все ще дуже ефективно.

– Особливість: Головна перевага — елементи в дереві завжди зберігаються у відсортованому порядку. Ітерація по такій множині завжди повертатиме елементи впорядковано. Це також дозволяє ефективно виконувати операції, яких немає у `HashSet`, наприклад, “знайти найменший/найбільший елемент” або “знайти всі елементи в діапазоні”.

Операція	`HashSet` (Хеш-таблиця)	`TreeSet` (Збалансоване дерево)
`add`	$O(1)$ середній	$O(\log n)$
`remove`	$O(1)$ середній	$O(\log n)$
`contains`	$O(1)$ середній	$O(\log n)$
Впорядкований обхід	Неможливий	$O(n)$

Мультимножина (Multiset) та Мішок (Bag)

Мультимножина (Multiset)

- Ідея: Що робити, якщо нам потрібно зберігати елементи без порядку, але при цьому дозволити дублікати? Наприклад, порахувати, скільки разів кожне слово зустрічається в тексті. Для цього існує мультимножина.
- Визначення: Це невпорядкована колекція елементів, яка дозволяє дублікати і зазвичай зберігає лічильник для кожної унікальної одиниці.
- Аналогія: Мішок з мармуровими кульками різних кольорів. Ви можете мати три червоні, п’ять синіх і одну зелену кульку.
- Операції: Інтерфейс схожий на множину, але з відмінностями:
  - **`add(item)`**: Якщо елемент вже є, його внутрішній лічильник збільшується на 1. Якщо ні — додається з лічильником 1.
  - **`remove(item)`**: Зменшує лічильник на 1. Якщо лічильник стає 0, елемент видаляється повністю.
  - **`count(item)`**: Нова операція, що повертає кількість входжень елемента.
- Реалізація: Зазвичай реалізується на основі хеш-таблиці, що відображає елемент на його лічильник (наприклад, **`HashMap<T, Integer>`**).

Мішок (Bag) У більшості контекстів комп’ютерних наук терміни “Мішок” (Bag) та “Мультимножина” (Multiset) використовуються як синоніми для позначення невпорядкованої колекції з можливими дублікатами.

Область застосування

- Множина (Set): Використовується скрізь, де потрібна унікальність: відстеження унікальних відвідувачів сайту, зберігання набору активних з'єднань, пошук унікальних слів у тексті, представлення множини вершин у графових алгоритмах.
- Мультимножина (Multiset): Застосовується там, де важлива не лише наявність, а й кількість: підрахунок частоти слів, аналіз інвентарю товарів на складі, представлення кошика покупця в інтернет-магазині, де може бути кілька одиниць одного товару.

Приклади застосування

- Пошук унікальних IP-адрес: Веб-сервер може використовувати `HashSet` для зберігання IP-адрес відвідувачів за добу, щоб швидко підрахувати кількість унікальних користувачів. `add` операція автоматично ігноруватиме дублікати.
- Підрахунок голосів: Мультимножина ідеально підходить для підрахунку голосів на виборах. Кожен голос за кандидата додається до мультимножини, а в кінці операція `count` для кожного кандидата покаже фінальний результат.

Переваги та недоліки

Переваги

- Швидка перевірка належності: Основна перевага `HashSet` — це можливість перевірити наявність елемента за середній час $O(1)$.
- Гарантія унікальності: Структура даних сама дбає про те, щоб не було дублікатів.
- Зручні теоретико-множинні операції: Легко виконувати об'єднання, перетин та різницю.

Недоліки

- Втрата порядку: Більшість ефективних реалізацій (на хеш-таблицях) не зберігають порядок додавання елементів.
- Втрата інформації про кількість: Класична множина не може зберігати інформацію про те, скільки разів елемент був доданий. Для цього потрібна мультимножина.

Контрольні запитання

3. У чому полягає ключова відмінність між АТД “Список” та АТД “Множина”?
4. Порівняйте реалізації множини `HashSet` та `TreeSet`. У якій ситуації ви б обрали `TreeSet`, незважаючи на його гіршу асимптотичну складність операцій?
5. Наведіть приклад задачі, де використання “Множини” є недоцільним, а “Мультимножини” — необхідним.
6. Опишіть, як би ви реалізували операцію `intersection(Set A, Set B)` для двох множин, реалізованих на хеш-таблицях. Якою буде її складність?
7. Що таке аксіоми АТД і чому вони важливі? Наведіть приклад однієї аксіоми для АТД “Множина”.

Методичні вказівки до лабораторної роботи

1 Назва

Реалізація АТД “Множина” (`HashSet`) на основі хеш-таблиці з методом ланцюжків.

2. Мета

Набути практичних навичок у реалізації множини. Зрозуміти, як хеш-функції та механізми вирішення колізій використовуються для забезпечення ефективних операцій.

3. Завдання

Створити клас `HashSet`, який реалізує АТД “Множина” для зберігання рядків. Для вирішення колізій використовувати метод ланцюжків (масив зв’язаних списків).

4. Опис.

Клас `HashSet` повинен мати приватне поле — масив, де кожен елемент є вказівником на голову зв’язного списку. Також клас повинен містити хеш-функцію для перетворення рядків на індекси масиву.

5. Порядок виконання

1. Опишіть структуру: Створіть клас `HashSet` та внутрішню структуру для вузла зв’язаного списку.

2. Реалізуйте хеш-функцію: Напишіть просту хеш-функцію для рядків (наприклад, на основі суми кодів символів за модулем розміру масиву).
3. Ініціалізуйте таблицю: В конструкторі створіть масив вказівників та ініціалізуйте їх `NULL`.
4. Реалізуйте `add(item)`:
    - Обчисліть хеш-індекс для `item`.
    - Перевірте, чи `item` вже присутній у зв'язному списку за цим індексом.
    - Якщо ні, додайте новий вузол на початок цього списку.
5. Реалізуйте `contains(item)`: Обчисліть хеш-індекс та виконайте пошук у відповідному зв'язному списку.
6. Реалізуйте `remove(item)`: Знайдіть та видаліть відповідний вузол зі зв'язного списку.
7. Протестуйте: Створіть об'єкт, додайте декілька елементів, у тому числі ті, що викликають колізії, перевірте наявність та видаліть деякі з них.

6. Результат

Повністю функціональний клас `HashSet`, що коректно виконує основні операції множини.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією класу `HashSet`.
2. Продемонструвати роботу додатку: показати додавання, видалення та перевірку наявності елементів, включаючи випадки колізій.
3. Бути готовим відповісти на питання: .....
    - Пояснити, як працює ваша хеш-функція.
    - Як ваш код обробляє колізії?
    - Яка складність операцій у вашій реалізації в найкращому, середньому та найгіршому випадках?

Тести

1. Яка основна властивість АТД "Множина"?

а) Елементи зберігаються у відсортованому порядку.

b) Всі елементи є унікальними.

c) Доступ до елемента здійснюється за індексом.

d) Працює за принципом LIFO.

2. Яку реалізацію множини варто обрати, якщо потрібно часто отримувати елементи у відсортованому порядку?

a) `HashSet`

b) `TreeSet`

c) На основі масиву без сортування.

d) На основі стеку.

3. У чому полягає головна відмінність Мультимножини від Множини?

a) Мультимножина зберігає порядок додавання.

b) Мультимножина не дозволяє видаляти елементи.

c) Мультимножина є швидшою.

d) Мультимножина дозволяє зберігати дублікати елементів.

4. Яка середня часова складність операції `contains` для реалізації `HashSet`?

a) $O(1)$

b) $O(\log n)$

c) $O(n)$

d) $O(n^2)$

5. Яка теоретико-множинна операція знаходить елементи, що присутні в обох множинах одночасно?

a) Об’єднання (Union)

b) Різниця (Difference)

c) Перетин (Intersection)

d) Декартів добуток

Відповіді: 1-b, 2-b, 3-d, 4-a, 5-c

Тести підвищеної складності

1. Ви реалізуєте систему, де потрібно зберігати набір унікальних ID користувачів та дуже часто виконувати операцію “знайти наступний за величиною ID” (наприклад, для ітерації по користувачах

впорядковано). Яка реалізація АТД “Множина” буде найбільш ефективною для всіх вимог задачі?

a) HashSet (на основі хеш-таблиці), оскільки вона забезпечує найшвидший середній час пошуку елемента.

b) TreeSet (на основі збалансованого дерева), оскільки, крім ефективного пошуку ($O(\log n)$), вона підтримує порядок, що робить пошук наступного елемента також дуже швидким ($O(\log n)$).

c) Множина на основі динамічного масиву з бінарним пошуком, оскільки це найпростіша впорядкована структура.

d) HashSet з додатковим зв’язним списком, що підтримує порядок вставки.

2. Розглянемо порожню мультимножину (Multiset/Bag), яка дозволяє зберігати дублікати. Виконується наступна послідовність операцій: `add(10)`, `add(20)`, `add(10)`, `add(30)`, `add(10)`, `remove(20)`. Якими будуть фінальні значення `size()` (загальна кількість елементів) та `count(10)` (кількість входжень елемента 10)?

a) `size()` = 3, `count(10)` = 1

b) `size()` = 4, `count(10)` = 3

c) `size()` = 3, `count(10)` = 3

d) `size()` = 4, `count(10)` = 2

3. Яка часова складність є найбільш ефективною для обчислення перетину (intersection) двох множин (Set), реалізованих на основі хеш-таблиць (HashSet), розмірами M та N (де M < N)?

a) $O(M \cdot N)$, оскільки потрібно порівняти кожен елемент однієї множини з кожним елементом іншої.

b) $O(N \log M)$, оскільки потрібно для кожного елемента більшої множини робити двійковий пошук у меншій.

c) $O(M)$, оскільки можна ітерувати по меншій множині та перевіряти наявність кожного її елемента в більшій множині за час $O(1)$.

d) $O(M + N)$, оскільки потрібно пройти по обох множинах.

4. У чому полягає ключова відмінність між Мультимножиною (Multiset) та Мішком (Bag), згідно з класичними визначеннями в інформатиці?

a) Мультимножина зберігає порядок елементів, а Мішок — ні.

b) Мішок дозволяє зберігати дублікати, а Мультимножина — ні.

c) Жодної різниці, це повні синоніми.

d) Мультимножина зазвичай реалізується на дереві та є впорядкованою, тоді як Мішок — це більш загальне поняття невпорядкованої колекції з дублікатами, часто реалізоване на хеш-таблиці.

5. Ви реалізуєте систему кешування, де потрібно зберігати до N унікальних об'єктів та при переповненні кешу видаляти той, до якого зверталися найдавніше (LRU - Least Recently Used). Яка комбінація структур даних буде найефективнішою для цього?

a) Одна хеш-таблиця, де значення зберігають час останнього доступу.

b) Двозв'язний список, відсортований за часом доступу.

c) Черга з пріоритетами, де пріоритет — це час доступу.

d) Комбінація хеш-таблиці (для доступу за ключем за $O(1)$) та двозв'язного списку (для швидкого переміщення елемента в кінець/видалення з початку за $O(1)$).

Відповіді

1. b) Хоча HashSet забезпечує пошук за $O(1)$, вона не зберігає порядок, тому операція "знайти наступний" вимагатиме перегляду всіх елементів ($O(n)$). TreeSet зберігає елементи у збалансованому дереві, що дозволяє знаходити як сам елемент, так і наступний за ним за логарифмічний час $O(\log n)$.

2. b) `size()` = 4, `count(10)` = 3

Трасування:

1. `add(10)`: `{10}`
2. `add(20)`: `{10, 20}`
3. `add(10)`: `{10, 10, 20}`
4. `add(30)`: `{10, 10, 20, 30}`
5. `add(10)`: `{10, 10, 10, 20, 30}`
6. `remove(20)`: `{10, 10, 10, 30}` *Фінальний стан:* Загальна кількість елементів (`size`) — 4. Кількість входжень `10` (`count(10)`) — 3.

3. c) $O(M)$ Найефективніший підхід — ітерувати по меншій множині (M елементів) і для кожного її елемента перевіряти, чи він присутній у більшій

множині. Оскільки перевірка наявності в HashSet займає в середньому $O(1)$, загальна складність буде $M \cdot O(1) = O(M)$.

4. d) Хоча терміни часто використовуються як взаємозамінні, в академічному сенсі та в багатьох стандартних бібліотеках (напр., C++ STL) Multiset — це впорядкована колекція з дублікатами (реалізована на дереві), а Bag — це більш загальне математичне поняття невпорядкованої колекції з дублікатами, яке найчастіше реалізують через хеш-таблицю, що зберігає пари (елемент, лічильник).

5. d) Це класична реалізація LRU кешу. Хеш-таблиця надає миттєвий доступ до вузла зв'язного списку за ключем. Двозв'язний список дозволяє миттєво перемістити вузол, до якого звернулися, в кінець (позначивши його як "останній використаний") та миттєво видалити вузол з початку ("найдавніший використаний"), коли кеш переповнено.

Завдання для самостійної роботи

1. Реалізація **`TreeSet`**. Реалізуйте спрощений АТД "Множина" на основі збалансованого бінарного дерева пошуку (можна використати невибагливу реалізацію BST без балансування для навчальних цілей).
2. Реалізація **`Multiset`**. Реалізуйте клас **`Multiset`** на основі хеш-таблиці, що відображає елементи на їхні лічильники.
3. Теоретико-множинні операції. Додайте до вашої реалізації **`HashSet`** методи для обчислення об'єднання, перетину та різниці двох множин.
4. Унікальні символи. Напишіть функцію, яка використовує множину для ефективного підрахунку кількості унікальних символів у довгому рядку.
5. Пошук дублікатів. Напишіть функцію, що приймає масив цілих чисел і повертає **`true`**, якщо він містить дублікати, і **`false`** в іншому випадку. Розв'яжіть задачу за допомогою множини. Яка складність такого розв'язку?
6. Індекс Жаккара. Напишіть функцію, що обчислює індекс подібності Жаккара для двох множин (розмір перетину, поділений на розмір об'єднання).
7. Перевірка на підмножину. Реалізуйте метод **`isSubset(otherSet)`**, що перевіряє, чи є одна множина підмножиною іншої.

8. Частотний аналіз тексту. Використовуючи вашу реалізацію `Multiset`, напишіть програму, яка аналізує текстовий файл і виводить 10 найуживаніших слів.
9. Видалення анаграм. Вам дано список слів. Напишіть функцію, що видаляє зі списку всі слова, які є анаграмами раніше зустрінутих слів (напр., зі списку `["listen", "silent", "enlist", "hello"]` залишиться `["listen", "hello"]`).
10. Есе-порівняння. Напишіть есе, що порівнює ефективність розв'язання задачі "знайти дублікати в масиві" трьома способами: повним перебором ($O(n^2)$), сортуванням ($O(n\log n)$) та використанням хеш-множини ($O(n)$).

Частина II: Алгоритми пошуку

Лінійні алгоритми пошуку

Алгоритми пошуку — це фундамент, з якого починається вивчення багатьох більш складних тем.

Окрім лінійного пошуку, який є найпростішим методом, існує декілька інших фундаментальних алгоритмів, кожен з яких має свої переваги та умови застосування.

Перелічимо основні з них:

Лінійний пошук.

- Ідея: Послідовний перегляд кожного елемента в колекції, доки не буде знайдено потрібний елемент або доки не закінчаться елементи.
- Вимоги до даних: Жодних. Працює на будь-яких, навіть невпорядкованих, даних.
- Складність: $O(n)$ у найгіршому випадку.
- Коли використовувати: Коли дані не відсортовані або їхня кількість дуже мала.

Бінарний пошук.

- Ідея: На кожному кроці масив ділиться навпіл. Алгоритм порівнює шуканий елемент із серединним і відкидає ту половину, в якій елемента гарантовано не може бути. Процес повторюється для половини, що залишилась.
- Вимоги до даних: Масив має бути відсортованим.
- Складність: $O(\log n)$. Це надзвичайно ефективний алгоритм.
- Коли використовувати: Це стандартний вибір для пошуку у великих відсортованих масивах.

Пошук стрибками.

- Ідея: Покращення лінійного пошуку для відсортованих масивів. Замість перегляду кожного елемента, алгоритм “стрибає” вперед на фіксований крок (зазвичай $\sqrt{n}$), доки не “перестрибне” шуканий елемент. Після цього виконується лінійний пошук у блоці між попереднім та поточним стрибком.
- Вимоги до даних: Масив має бути відсортованим.

– Складність: $O(\sqrt{n})$. Швидше, ніж лінійний пошук, але повільніше, ніж двійковий.

– Коли використовувати: На практиці рідко, але може бути корисним у системах, де стрибок назад є значно “дорожчим”, ніж стрибок уперед.

Інтерполяційний пошук.

– Ідея: Покращення двійкового пошуку. Замість того, щоб завжди перевіряти середній елемент, він робить “розумне припущення” про місцезнаходження елемента, базуючись на його значенні. Аналогічно до того, як ви шукаєте прізвище на літеру “Т” ближче до кінця телефонної книги, а не посередині.

– Вимоги до даних: Масив має бути відсортованим та рівномірно розподіленим.

– Складність: $O(\log(\log n))$ в середньому, але у найгіршому випадку (при нерівномірному розподілі) може впасти до $O(n)$.

– Коли використовувати: Для великих, відсортованих та рівномірно розподілених наборів даних, як-от телефонні довідники.

Експоненційний пошук

– Ідея: Використовується для пошуку в необмежених (або дуже великих) відсортованих масивах. Спочатку алгоритм знаходить діапазон, в якому може знаходитись елемент, перевіряючи індекси $1,2,4,8,16,\ldots,2^k$, доки елемент на цій позиції не стане більшим за шуканий. Після цього виконується двійковий пошук у знайденому діапазоні [2^{k-1}, 2^k].

– Вимоги до даних: Масив має бути відсортованим.

– Складність: $O(\log n)$.

– Коли використовувати: Коли масив занадто великий, щоб поміститися в пам’ять, або його розмір невідомий.

Ці алгоритми ми й розглянемо більш детально.

Лінійний пошук

Теоретичні відомості

Уявіть, що перед вами на столі лежить стопка карток з іменами, і вам потрібно знайти картку з ім’ям “Олександр”. Картки лежать у довільному порядку. Якою буде ваша найбільш очевидна стратегія? Ви візьмете верхню

картку, подивитесь на неї. Якщо це не “Олександр”, ви відкладете її і візьмете наступну. Ви будете повторювати цей процес, доки не знайдете потрібну картку або доки картки не закінчаться.

Цей простий, інтуїтивно зрозумілий процес і є лінійним (або послідовним) пошуком.

Лінійний пошук — це фундаментальний алгоритм, який знаходить потрібний елемент у колекції (наприклад, у масиві) шляхом послідовного перегляду кожного елемента, один за одним, від початку до кінця, доки не буде знайдено збіг.

Це метод “грубої сили” у світі пошуку: він не робить жодних припущень щодо впорядкованості даних і просто перевіряє всі можливості. Його головна сила — в його універсальності. Він буде працювати для будь-якого списку, який ви йому дасте.

Псевдокод та його аналіз

Давайте формалізуємо цей процес у вигляді псевдокоду. Наша функція буде приймати на вхід масив `A` та ключ `key`, який ми шукаємо. Якщо елемент знайдено, функція поверне його індекс. Якщо ні — поверне спеціальне значення, наприклад, -1, що сигналізує про невдачу.

```
// Функція лінійного пошуку
// Вхід: A - масив даних, key - значення, яке потрібно знайти.
// Вихід: індекс першого входження key в A, або -1, якщо key не знайд
ено.
ФУНКЦІЯ LinearSearch(Array A, key):
    // Отримуємо кількість елементів у масиві.
    n = довжина(A)

    // Запускаємо цикл, що пройде по кожному індексу від 0 до n-1.
    ДЛЯ i ВІД 0 ДО n-1:
        // На кожній ітерації порівнюємо поточний елемент A[i] з ключ
ем.
        ЯКЩО A[i] == key ТО
            // Якщо знайдено збіг, негайно повертаємо поточний індекс
.
            ПОВЕРНУТИ i
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    // Якщо цикл завершився, а ми так і не повернули індекс,
    // це означає, що елемент у масиві відсутній.
```

```
    ПОВЕРНУТИ -1
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності: найкращий, середній та найгірший випадки

Незважаючи на простоту, дуже важливо проаналізувати ефективність цього алгоритму.

- Найкращий випадок: Нам неймовірно пощастило, і елемент, який ми шукаємо, знаходиться на самому початку масиву, на позиції з індексом 0. Алгоритм виконає лише одне порівняння і одразу завершить роботу. Часова складність: $O(1)$ (константний час).
- Найгірший випадок: У цьому сценарії нам не щастить. Елемент, який ми шукаємо, знаходиться в самому кінці масиву, або його взагалі немає в масиві. В обох ситуаціях алгоритму доведеться пройти через усі n елементів, виконавши n порівнянь. Часова складність: $O(n)$ (лінійний час). Саме на цей показник зазвичай орієнтуються при оцінці алгоритму.
- Середній випадок: Припустимо, що шуканий елемент гарантовано є в масиві і може з однаковою ймовірністю знаходитись на будь-якій позиції. Тоді в середньому нам доведеться перевірити половину масиву, тобто *n/2* елементів. В асимптотичному аналізі ми відкидаємо константні множники (в даному випадку, 1/2), тому середня складність також є лінійною. Часова складність: $O(n)$.
- Просторова складність: Важливо відзначити, що лінійний пошук є дуже економним з точки зору пам'яті. Для своєї роботи він використовує лише декілька змінних (наприклад, лічильник циклу **`i`**), і ця кількість не залежить від розміру вхідного масиву. Просторова складність: $O(1)$ (константний час).

Область застосування

Хоча лінійний пошук є асимптотично повільним для великих наборів даних, існують сценарії, де він є не тільки прийнятним, але й найкращим вибором:

- Малі колекції: Якщо вам потрібно знайти щось у масиві з 10, 20 або навіть 100 елементів, лінійний пошук виконається практично миттєво. Складність реалізації та накладні витрати більш просунутих алгоритмів будуть невиправданими.

– Невпорядковані дані: Це головний козир лінійного пошуку. Якщо дані не відсортовані, такі потужні методи, як бінарний пошук, просто неможливо застосувати.
– Одноразовий пошук: Якщо вам потрібно виконати пошук лише один раз, то витрачати час на попереднє сортування даних ($O(n\log n)$) для того, щоб потім зробити один швидкий пошук ($O(\log n)$), немає сенсу. Простіше одразу виконати один лінійний пошук за $O(n)$.
– Пошук у зв'язаних списках: Оскільки зв'язані списки не надають доступу до елемента за індексом за $O(1)$, єдиний спосіб знайти в них елемент — це послідовно перебирати вузли від голови, що є, по суті, лінійним пошуком.

Приклади застосування

– Пошук товару в чеку: Коли ви скануєте чек, щоб знайти, скільки коштував хліб, ви послідовно переглядаєте кожен рядок, доки не знайдете потрібний.
– Перевірка наявності у списку запрошених: Охоронець на вході до клубу переглядає список імен по черзі, щоб перевірити, чи є ви у списку.
– Прості фільтри в програмах: Коли ви вводите текст у поле "фільтр" в налаштуваннях, програма часто просто проходить по всіх доступних опціях і показує ті, що містять введений вами текст.

Переваги та недоліки

Переваги:

- Простота: Алгоритм надзвичайно простий для розуміння, реалізації та налагодження.
- Універсальність: Працює на будь-яких колекціях (масиви, списки) незалежно від того, чи вони відсортовані.
- Не потребує підготовки: Немає потреби у попередньому сортуванні чи будь-якій іншій обробці даних.
- Мінімальні вимоги до пам'яті ($O(1)$).

Недоліки:

- Неефективність: Для великих обсягів даних (n 10,000) стає дуже повільним.

- Погано масштабується: Збільшення обсягу даних вдвічі призводить до збільшення середнього часу пошуку вдвічі.

Контрольні запитання

1. У чому полягає основна ідея лінійного пошуку?
2. Опишіть найкращий та найгірший сценарії для лінійного пошуку. Якою буде складність у кожному з цих випадків?
3. Чому, незважаючи на існування значно швидших алгоритмів, лінійний пошук все ще використовується на практиці? Наведіть два приклади.
4. Чому для пошуку у зв'язаному списку ми змушені використовувати підхід, аналогічний лінійному пошуку, навіть якщо список відсортований?
5. Порівняйте часову та просторову складність лінійного пошуку.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та аналіз алгоритму лінійного пошуку.

2. Мета

Набути практичних навичок у реалізації базового алгоритму пошуку. Емпірично дослідити його поведінку в найкращому, середньому та найгіршому випадках.

3. Завдання

Створити програму, яка реалізує функцію лінійного пошуку та демонструє її роботу на масиві цілих чисел.

4. Опис

Програма має містити функцію `LinearSearch`, яка приймає масив та ключ. Головна частина програми повинна створювати тестовий масив та викликати цю функцію для різних сценаріїв пошуку.

6. Порядок виконання

    1. Реалізуйте функцію: Напишіть функцію `LinearSearch`, керуючись наведеним псевдокодом.
    2. Створіть тестовий масив: В основній частині програми створіть масив з 10-15 цілих чисел (наприклад, `{5, 12, 3, 88, 42, ...}`).

3. Перевірте найкращий випадок: Викличте вашу функцію для пошуку першого елемента масиву. Виведіть результат.
4. Перевірте найгірший випадок: Викличте вашу функцію для пошуку останнього елемента масиву. Виведіть результат.
5. Перевірте випадок відсутності елемента: Викличте вашу функцію для пошуку числа, якого гарантовано немає в масиві. Переконайтесь, що функція повертає -1.
6. Зробіть висновки: Проаналізуйте отримані результати.

7. Результат

Додаток, що коректно знаходить елементи в масиві за допомогою лінійного пошуку та обробляє випадок, коли елемент відсутній.

9. Після виконання лабораторної роботи студент повинен надати:
    1. Вихідний код програми з реалізацією функції `LinearSearch` та тестовими викликами.
    2. Продемонструвати роботу додатку, показавши результати для всіх трьох сценаріїв (найкращий, найгірший, відсутній елемент).
8. Бути готовим відповісти на питання:
    - Пояснити кожний рядок вашої реалізації.
    - Яка складність вашого алгоритму?
    - Що змінилося б, якби ваш масив був відсортований? Чи став би лінійний пошук працювати швидше?

Тести

1. Яка основна вимога до даних для застосування лінійного пошуку?

a) Дані мають бути відсортовані.

b) Дані мають бути цілими числами.

c) Дані мають бути унікальними.

d) Жодних спеціальних вимог.

2. У масиві з 1,000,000 елементів ми шукаємо елемент, що знаходиться на першій позиції. Скільки порівнянь виконає лінійний пошук?

a) 1,000,000

b) 500,000

c) 1

d) $\log_2(1{,}000{,}000)$

3. Яка просторова складність лінійного пошуку?

a) $O(n)$

b) $O(1)$

c) $O(\log n)$

d) $O(n^2)$

4. В якому випадку лінійний пошук є кращим вибором, ніж бінарний?

a) Коли масив дуже великий.

b) Коли дані не відсортовані.

c) Коли всі елементи масиву однакові.

d) Такого випадку не існує.

7. Часова складність лінійного пошуку у найгіршому випадку описується як:

a) $O(1)$

b) $O(\log n)$

c) $O(n)$

d) $O(n \log n)$

Відповіді: 1-d, 2-c, 3-b, 4-b, 5-c

Тести підвищеної складності

1. Якщо припустити, що шуканий елемент гарантовано існує в масиві з N унікальних елементів і ймовірність його знаходження на будь-якій позиції однакова, яким буде очікуване (середнє) число порівнянь для успішного пошуку?

a) N

b) N / 2

c) (N + 1) / 2

d) $\log_2 N$

2. У якому з наступних сценаріїв використання пошуку з бар'єром (Sentinel Search) може виявитись повільнішим за стандартний лінійний пошук, навіть якщо обидва алгоритми виконуються багато разів?

a) Коли масив містить велику кількість дублікатів шуканого елемента.

b) Коли масив дуже малий (наприклад, 3-5 елементів), і вартість операцій збереження та відновлення останнього елемента перевищує виграш від усунення однієї перевірки в циклі.

c) Коли масив відсортований у зворотному порядку.

d) Коли пошук виконується у функції, що викликається рекурсивно.

3. Вам потрібно реалізувати лінійний пошук для структури даних, що зберігає мільйон елементів. На якій структурі даних, за однакових умов, пошук буде найповільнішим на практиці, незважаючи на однакову асимптотичну складність $O(N)$?

a) Динамічний масив (вектор).

b) Статичний масив.

c) Однозв'язний список.

d) Двозв'язний список.

4. Використовуючи лінійний пошук, вам потрібно знайти два різних індекси `i` та `j` в масиві `A`, для яких `A[i] + A[j] == sum`. Яка часова складність найпростішого "лобового" розв'язку?

a) $O(N)$

b) $O(N\log N)$

c) $O(N^2)$

d) $O(\log N)$

5. Ви модифікували лінійний пошук так, щоб він перевіряв елементи у випадковому порядку, а не послідовно. Як це вплине на часову складність у найгіршому випадку?

a) Складність покращиться до $O(\log N)$, оскільки випадковий вибір може швидше знайти елемент.

b) Складність залишиться $O(N)$, оскільки в найгіршому випадку все одно доведеться перевірити всі елементи, хоч і в іншому порядку.

c) Складність погіршиться до $O(N^2)$, оскільки генерація випадкових індексів є дорогою операцією.

d) Складність стане $O(1)$, оскільки є шанс знайти елемент з першої спроби.

Відповіді

1. c) (N + 1) / 2 Щоб знайти перший елемент, потрібне 1 порівняння, другий — 2, і т.д., до N-го, для якого потрібно N порівнянь. Середня кількість порівнянь — це сума арифметичної прогресії `(1 + 2 + ... + N) / N = (N * (N + 1) / 2) / N = (N + 1) / 2`. Для великих N це наближається до N/2, але (N+1)/2 є більш точною відповіддю.

2. b) На дуже малих масивах фіксовані накладні витрати на збереження (`last_element = A[n-1]`), запис "вартового" (`A[n-1] = key`), відновлення (`A[n-1] = last_element`) та фінальну перевірку (`if...`) можуть "з'їсти" весь виграш від усунення однієї перевірки в тілі дуже короткого циклу.

3. c)та d) Однозв'язний та двозв'язний списки будуть найповільнішими. На відміну від масиву, де елементи зберігаються в суцільному блоці пам'яті (що добре для кешу процесора), вузли зв'язного списку розкидані по пам'яті. Кожен перехід до наступного елемента (`->next`) з високою ймовірністю призводить до кеш-промаху (cache miss) і повільного звернення до основної оперативної пам'яті.

4. c) $O(N^2)$ Найпростіший розв'язок — це використання двох вкладених циклів. Зовнішній цикл бере елемент `A[i]`, а внутрішній проходить по решті елементів `A[j]`, перевіряючи умову `A[i] + A[j] == sum`. Це призводить до квадратичної складності.

5. b) Асимптотична складність у найгіршому випадку не зміниться. Найгірший випадок настає тоді, коли елемента немає в масиві. Незалежно від порядку перевірки (послідовно чи випадково), щоб у цьому переконатися, нам все одно доведеться переглянути всі N елементів.

Завдання для самостійної роботи

1. Пошук останнього входження. Модифікуйте алгоритм лінійного пошуку так, щоб він повертав індекс останнього входження елемента в масив, а не першого.
2. Підрахунок входжень. Напишіть функцію, яка використовує лінійний пошук для підрахунку кількості входжень заданого елемента в масив.
3. Пошук у 2D масиві. Реалізуйте функцію, що виконує лінійний пошук у двовимірній матриці та повертає координати (рядок, стовпець) знайденого елемента.

4. Лінійний пошук для зв’язного списку. Напишіть реалізацію лінійного пошуку для структури даних “однозв’язний список”.
5. Пошук за умовою. Напишіть функцію, яка приймає масив чисел і знаходить перший елемент, що задовольняє певній умові (наприклад, перше парне число, більше за 50).
6. “Вартовий” лінійний пошук. Дослідіть та реалізуйте оптимізовану версію лінійного пошуку, яка використовує “вартового”. Ідея полягає в тому, щоб додати шуканий елемент в кінець масиву, що дозволяє прибрати перевірку на вихід за межі масиву з тіла циклу. Порівняйте кількість операцій на кожній ітерації.
7. Пошук мінімуму та максимуму. Напишіть функцію, яка за один прохід по масиву (за один цикл) знаходить і мінімальний, і максимальний елементи.
8. Generic-пошук. Напишіть узагальнену версію лінійного пошуку, яка може працювати з масивом будь-якого типу даних (цілі числа, рядки, об’єкти), використовуючи передану функцію для порівняння.
9. Пошук першого унікального. Для заданого масиву символів знайдіть перший символ, який не повторюється. Проаналізуйте складність вашого розв’язку.
10. Емпіричне порівняння. Напишіть програму, що порівнює реальний час виконання лінійного пошуку на масиві та `std::list` (C++) або `LinkedList` (Java). Поясніть отримані результати.

Бінарний пошук

Ми продовжуємо наш шлях у світ ефективних алгоритмів. Після знайомства з універсальним, але повільним лінійним пошуком, ми розглянемо його значно швидшого та “розумнішого” колегу. Цей алгоритм є одним з найважливіших у комп’ютерних науках і є яскравим прикладом парадигми “Розділяй і володарюй”.

Теоретичні відомості

Ідея бінарного пошуку: відкидаючи половину всесвіту

У попередньому розділі ми встановили, що лінійний пошук має складність $O(n)$, оскільки в найгіршому випадку ми змушені переглянути всі елементи. Це надійно, але для великих обсягів даних — катастрофічно повільно.

Але що, якби ми мали додаткову інформацію про наші дані? Що, якби вони були відсортовані?

Це єдина, але абсолютно критична вимога для двійкового пошуку. Уявіть, що ви шукаєте слово “оптимізація” у великому паперовому словнику. Ви не будете гортати його з першої сторінки. Ви відкриєте його приблизно посередині, наприклад, на літері “М”. Ви миттєво розумієте, що “О” йде після “М”, тому вся перша половина словника вам більше не потрібна. Одним рухом ви щойно відкинули половину простору для пошуку. Далі ви берете другу половину і повторюєте той самий трюк — ділите її навпіл.

Цей процес — систематичне ділення впорядкованої колекції навпіл і відкидання непотрібної частини — і є суттю двійкового (або бінарного) пошуку.

Формальна ідея:

1. Беремо відсортований масив і визначаємо межі пошуку: початок (`left`) та кінець (`right`).
2. Знаходимо середній елемент (`middle`).
3. Порівнюємо шуканий ключ із середнім елементом.
    - Якщо вони збігаються — ми знайшли те, що шукали.
    - Якщо наш ключ менший за середній, ми можемо бути на 100% впевнені, що він, якщо і існує, знаходиться у лівій половині масиву. Ми відкидаємо праву половину, просто зсунувши межу `right`.
    - Якщо наш ключ більший за середній, він може бути лише у правій половині. Ми відкидаємо ліву, зсуваючи межу `left`.
4. Повторюємо кроки 2-3 для нової, зменшеної вдвічі, області пошуку, доки не знайдемо елемент або доки область пошуку не звузиться до нуля.

Ітеративна реалізація

Це класична і найбільш поширена реалізація двійкового пошуку, яка використовує цикл. Вона є трохи ефективнішою за рекурсивну, оскільки не має накладних витрат на виклики функцій.

Псевдокод:

```
// Вхід: A - відсортований масив, key - значення, яке потрібно знайти
.
```

```
// Вихід: індекс знайденого елемента, або -1, якщо не знайдено.
ФУНКЦІЯ BinarySearch_Iterative(Array A, key):
    // Ініціалізуємо межі пошуку: від першого до останнього індексу.
    left = 0
    right = довжина(A) - 1

    // Цикл триває, доки ліва межа не перетне праву.
    ПОКИ left <= right:
        // Знаходимо індекс середнього елемента.
        // Цей спосіб обчислення запобігає переповненню для дуже вели
ких масивів.
        middle = left + (right - left) / 2

        // Перевіряємо середній елемент
        ЯКЩО A[middle] == key ТО
            // Елемент знайдено!
            ПОВЕРНУТИ middle
        КІНЕЦЬ ЯКЩО

        // Якщо наш ключ менший за середній, відкидаємо праву половин
у.
        ЯКЩО A[middle] > key ТО
            right = middle - 1
        // Інакше, якщо наш ключ більший, відкидаємо ліву половину.
        ІНАКШЕ
            left = middle + 1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ

    // Якщо цикл завершився, а елемент не знайдено.
    ПОВЕРНУТИ -1
КІНЕЦЬ ФУНКЦІЇ
```

Рекурсивна реалізація

Бінарний пошук є яскравим прикладом парадигми “Розділяй і володарюй”, тому його дуже природно реалізувати за допомогою рекурсії. Логіка залишається тією ж, але замість циклу ми використовуємо рекурсивні виклики.

Псевдокод:

```
// Вхід: A - відсортований масив, key - ключ, left/right - поточні ме
жі пошуку.
ФУНКЦІЯ BinarySearch_Recursive(Array A, key, left, right):
    // Базовий випадок рекурсії: межі перетнулися, елемент не знайден
о.
```

```
    ЯКЩО left > right ТО
        ПОВЕРНУТИ -1
    КІНЕЦЬ ЯКЩО

    // Знаходимо середину
    middle = left + (right - left) / 2

    // Перевіряємо середній елемент
    ЯКЩО A[middle] == key ТО
        ПОВЕРНУТИ middle
    КІНЕЦЬ ЯКЩО

    // Рекурсивний крок: викликаємо функцію для відповідної половини.
    ЯКЩО A[middle] > key ТО
        ПОВЕРНУТИ BinarySearch_Recursive(A, key, left, middle - 1)
    ІНАКШЕ
        ПОВЕРНУТИ BinarySearch_Recursive(A, key, middle + 1, right)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності: сила логарифму

Головна перевага бінарного пошуку — його феноменальна швидкість на великих масивах.

– Часова складність: На кожному кроці ми ділимо розмір масиву для пошуку приблизно навпіл. Якщо початковий розмір масиву — n, то після першого кроку він стане $n/2$, після другого — $n/4$, після третього — $n/8$, і так далі, доки не дійдемо до 1. Кількість кроків k, необхідних для цього, визначається рівнянням $n/2^k = 1$, або $n = 2^k$. Звідси $k = \log_2 n$. Таким чином, часова складність двійкового пошуку — $O(\log n)$.

  Щоб зрозуміти, наскільки це швидко:

  - Для масиву з 1,000,000 елементів лінійний пошук у найгіршому випадку зробить 1,000,000 порівнянь.
  - Бінарний пошук зробить лише близько 20 порівнянь ($\log_2 1{,}000{,}000 \approx 19.9$).

– Просторова складність:

  - Ітеративна версія: Використовує лише декілька змінних (`left`, `right`, `middle`), їхня кількість не залежить від розміру масиву. Тому просторова складність — $O(1)$.

- Рекурсивна версія: Кожен рекурсивний виклик займає місце у стеку викликів. Глибина рекурсії дорівнює кількості кроків, тобто $\log n$. Тому просторова складність — $O(\log n)$.

2. Область застосування

Бінарний пошук є одним з найбільш використовуваних алгоритмів в інформатиці. Він застосовується скрізь, де є потреба у швидкому пошуку в великому, впорядкованому наборі даних.

- Бази даних: Пошук записів за проіндексованими полями.
- Словники та довідники: Пошук слів або значень.
- Системи контролю версій: Пошук конкретного коміту в історії змін (яка є впорядкованою за часом).
- Обчислювальна геометрія.
- Як основа для більш складних алгоритмів.

Приклади застосування

- Пошук в телефонному довіднику: Коли ви шукаєте контакт у своєму смартфоні, система, швидше за все, використовує двійковий пошук по відсортованому списку контактів.
- Гра “Вгадай число”: Це пряма реалізація двійкового пошуку.
- Відлагодження коду (`git bisect`): У системі Git команда `bisect` використовує двійковий пошук по історії комітів, щоб швидко знайти той, який вніс помилку.

Переваги та недоліки

Переваги бінарного пошуку очевидні. Його логарифмічна часова складність $O(\log n)$ робить його надзвичайно швидким та ефективним для великих наборів даних. Він є значно продуктивнішим за лінійний пошук і простий в реалізації.

Однак, є і недоліки, а точніше, одна головна вимога, яка є його “ахіллесовою п’ятою”. Алгоритм працює виключно на відсортованих даних. Це означає, що якщо ваша колекція даних часто змінюється (додаються або видаляються елементи), вам доведеться постійно підтримувати її у відсортованому стані. Операція вставки елемента у відсортований масив зі збереженням порядку має складність $O(n)$, що може звести нанівець усі переваги швидкого пошуку.

Таким чином, бінарний пошук є ідеальним для статичних або рідко змінюваних наборів даних, де операція пошуку виконується набагато частіше, ніж операції модифікації.

Контрольні запитання

1. Яка найголовніша вимога до даних для застосування бінарного пошуку?
2. Поясніть своїми словами, чому складність бінарного пошуку є логарифмічною.
3. Опишіть, як змінюються межі `left` та `right` в ітеративному алгоритмі, якщо шуканий елемент виявляється більшим за середній.
4. Чому просторова складність рекурсивної версії бінарного пошуку відрізняється від ітеративної?
5. Наведіть приклад задачі, де, незважаючи на те, що дані можна відсортувати, використання бінарного пошуку буде недоцільним.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та порівняння ітеративної та рекурсивної версій бінарного пошуку.

2. Мета

Практичне засвоєння логіки алгоритму бінарного пошуку. Розуміння відмінностей між ітеративним та рекурсивним підходами до розв’язання однієї і тієї ж задачі.

3. Завдання

Створити програму, яка реалізує дві функції бінарного пошуку (ітеративну та рекурсивну) та демонструє їхню роботу на відсортованому масиві цілих чисел.

4. Опис

Програма повинна містити дві окремі функції, що реалізують алгоритм. Головна частина програми має створювати тестовий масив, сортувати його (якщо він не відсортований) і викликати обидві функції для демонстрації їхньої роботи.

5. Порядок виконання

1. Створіть тестовий масив: В основній частині програми створіть відсортований масив з 15-20 цілих чисел.
2. Реалізуйте ітеративну функцію: Напишіть функцію `BinarySearch_Iterative`, керуючись наведеним псевдокодом.
3. Реалізуйте рекурсивну функцію: Напишіть функцію `BinarySearch_Recursive`.
4. Протестуйте: Викличте обидві функції для різних сценаріїв:
    - Пошук елемента, що є в середині масиву.
    - Пошук першого елемента масиву.
    - Пошук останнього елемента масиву.
    - Пошук елемента, якого немає в масиві.
5. Виведіть результати: Для кожного сценарію виводьте на екран результат роботи обох функцій, щоб переконатись, що вони збігаються.

6. Результат

Додаток, що коректно знаходить елементи у відсортованому масиві за допомогою двох реалізацій бінарного пошуку.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією обох версій алгоритму.
2. Продемонструвати роботу додатку, показавши результати для всіх тестових сценаріїв.
3. Бути готовим відповісти на питання:
    - Пояснити, як працює базовий випадок у вашій рекурсивній реалізації.
    - Чому формула `middle = left + (right - left) / 2` є кращою за `middle = (left + right) / 2`?
    - Яка версія, на вашу думку, є кращою для практичного використання і чому?

Тести

1. Бінарний пошук можна застосувати до:

a) Будь-якого масиву.

b) Відсортованого масиву.

c) Зв’язного списку.

d) Хеш-таблиці.

2. Яка часова складність бінарного пошуку в найгіршому випадку?

a) $O(n)$

b) $O(1)$

c) $O(n^2)$

d) $O(\log n)$

3. Якщо `key > A[middle]`, як зміняться межі пошуку на наступному кроці?

a) `right = middle - 1`

b) `left = middle + 1`

c) `left = middle`

d) `right = middle`

4. Скільки приблизно порівнянь зробить бінарний пошук у масиві з 1000 елементів?

a) 1000

b) 500

c) 100

d) 10

5. Яка просторова складність ітеративної версії бінарного пошуку?

a) $O(n)$

b) $O(\log n)$

c) $O(1)$

d) $O(n^2)$

Відповіді: 1-b, 2-d, 3-b, 4-d (бо $2^{10} = 1024$), 5-c

Тести підвищеної складності

1. Чому в реалізаціях двійкового пошуку для мов програмування з фіксованим розміром цілих чисел (`int`) рекомендується використовувати формулу

   ```
   middle = left + (right - left) / 2
   ```

   замість

`middle = (left + right) / 2`?

a) Перша формула працює швидше, оскільки містить більше операцій.

b) Щоб уникнути помилки цілочисельного переповнення, коли `left` та `right` є дуже великими додатними числами, і їхня сума може перевищити максимальне значення `int`.

c) Друга формула не працює коректно з від'ємними індексами, а перша — працює.

d) Перша формула дає більш точний результат при роботі з нецілими індексами.

2. Вам дано відсортований масив, який циклічно зсунули на невідому кількість позицій (наприклад, `{6, 7, 8, 1, 2, 3, 4}`). Яка ключова ідея для адаптації двійкового пошуку, щоб знайти елемент за час $O(\log n)$?

a) Спочатку знайти точку зсуву (найменший елемент) за $O(n)$, а потім виконати двійковий пошук.

b) На кожному кроці, після знаходження `middle`, визначати, яка з двох половин (`[left...middle]` чи `[middle...right]`) є гарантовано відсортованою, і перевіряти, чи знаходиться ключ у ній.

c) Виконати два окремі бінарних пошуки: один для першої половини масиву, інший — для другої.

d) Використати інтерполяційний пошук, оскільки він краще працює на нелінійних даних.

3. Як потрібно модифікувати стандартний бінарний пошук, щоб гарантовано знайти індекс першого входження елемента `key` в масиві, де можуть бути дублікати?

a) Коли `A[middle] == key` знайдено, негайно зупинитись і повернути `middle`.

b) Коли `A[middle] == key` знайдено, перевірити `A[middle - 1]`. Якщо він менший, то `middle` — це перший.

c) Коли `A[middle] == key` знайдено, зберегти цей індекс як потенційний результат і продовжити пошук у правій половині (`left = middle + 1`), щоб знайти ще кращий варіант.

d) Коли `A[middle] == key` знайдено, зберегти цей індекс як потенційний результат і продовжити пошук у лівій половині (`right = middle - 1`), щоб знайти ще більш раннє входження.

4. При обчисленні цілочисельного квадратного кореня з числа `N` за допомогою бінарного пошуку, що є “відсортованим масивом”, по якому ми шукаємо?

a) Діапазон простих дільників числа `N`.

b) Послідовність бітів у двійковому представленні числа `N`.

c) Попередньо обчислений масив квадратів чисел.

d) Діапазон цілих чисел від `0` до `N`, де ми шукаємо таке число `x`, для якого `x*x` найбільш близьке до `N`.

5. У чому полягає ключова відмінність у припущеннях, які роблять бінарний пошук та інтерполяційний пошук, що призводить до їхньої різної середньої складності ($O(\log n)$ проти $O(\log(\log n))$)?

a) Бінарний пошук працює лише з цілими числами, а інтерполяційний — з числами з плаваючою комою.

b) Інтерполяційний пошук, на відміну від бінарного, не вимагає, щоб масив був відсортованим.

c) Інтерполяційний пошук додатково припускає, що дані в масиві розподілені рівномірно, що дозволяє йому робити більш точні “припущення” про позицію елемента.

d) Бінарний пошук завжди реалізується рекурсивно, а інтерполяційний — ітеративно.

Відповіді

1. b) Якщо `left` та `right` — це великі числа, близькі до максимального значення для `int` (напр., 2 мільярди), їхня сума `left + right` може перевищити це максимальне значення, спричинивши переповнення та некоректний результат. Формула `left + (right - left) / 2` математично еквівалентна, але уникає прямого додавання великих `left` та `right`.

2. b) Це основа розв’язку. На кожній ітерації одна з половин масиву відносно `middle` завжди буде нормально відсортована. Ми перевіряємо, чи знаходиться наш ключ у цій відсортованій частині. Якщо так — шукаємо в

ній. Якщо ні — шукаємо в іншій, “зламаній” частині. Це зберігає логарифмічну складність.

3. d) Щоб знайти перше входження, знайшовши збіг, ми не можемо зупинятись. Нам потрібно перевірити, чи не існує такого ж елемента ще лівіше. Тому ми фіксуємо поточний індекс як можливу відповідь і продовжуємо пошук у лівій половині.

4. d) Бінарний пошук можна застосувати до будь-якої впорядкованої множини, де можна визначити “менше”, “більше” або “дорівнює”. У цьому випадку ми шукаємо відповідь (`x`) у діапазоні можливих відповідей `[0...N]`. “Серединний” елемент, який ми перевіряємо, — це `middle*middle`.

5. c) Бінарний пошук не робить жодних припущень про розподіл даних, тому він “сліпо” перевіряє середину. Інтерполяційний пошук використовує інформацію про значення на кінцях діапазону, щоб зробити припущення про позицію елемента, і це припущення є точним лише для рівномірно розподілених даних.

Завдання для самостійної роботи

1. Пошук першого входження. Модифікуйте алгоритм бінарного пошуку так, щоб він знаходив індекс першого входження елемента в масив, де можуть бути дублікати.
2. Пошук останнього входження. Аналогічно, модифікуйте алгоритм для пошуку останнього входження елемента.
3. Кількість входжень. Використовуючи дві попередні функції, напишіть функцію, що за час $O(\log n)$ знаходить кількість входжень заданого елемента в масив.
4. Пошук у повернутому масиві. Вам дано відсортований масив, який циклічно зсунули на невідому кількість позицій (напр., `{4, 5, 6, 7, 0, 1, 2}`). Реалізуйте алгоритм пошуку елемента в такому масиві за $O(\log n)$.
5. Знаходження квадратного кореня. Реалізуйте функцію для обчислення цілочисельного квадратного кореня з числа N за допомогою двійкового пошуку по діапазону від 1 до N.

6. Пошук точки вставки. Напишіть функцію, яка для відсортованого масиву та нового елемента знаходить індекс, на який цей елемент можна було б вставити, не порушуючи порядку сортування.
7. Пошук найближчого елемента. Для відсортованого масиву та заданого числа `key` знайдіть елемент, найближчий до `key`.
8. Пошук пікового елемента. “Піковим” називається елемент, що більший за своїх сусідів. Знайдіть будь-який піковий елемент у масиві за час $O(\log n)$.
9. Бінарний пошук за відповіддю. Дослідіть цю техніку та розв’яжіть за її допомогою наступну задачу: є N кабелів різної довжини, потрібно розрізати їх так, щоб отримати K однакових шматків максимально можливої довжини.

Пошук з бар’єром

Розглянувши фундаментальні підходи, давайте зануримось у світ мікрооптимізацій. Іноді, навіть не змінюючи асимптотичну складність, можна зробити алгоритм помітно швидшим на практиці. Пошук з бар’єром — ідеальний приклад такої інженерної хитрості.

1. Теоретичні відомості

1.1. Повернення до лінійного пошуку: пошук прихованої неефективності

У попередніх розділах ми детально вивчили лінійний пошук. Його ідея проста — послідовно переглядати кожен елемент. Давайте ще раз подивимось на його псевдокод і проаналізуємо, що насправді відбувається всередині циклу.

```
ФУНКЦІЯ StandardLinearSearch(Array A, key):
    n = довжина(A)
    ДЛЯ і ВІД 0 ДО n-1:
        // На кожній ітерації виконується ДВІ перевірки:
        // 1. Чи поточний елемент є тим, що ми шукаємо?
        ЯКЩО A[i] == key ТО
            ПОВЕРНУТИ і
        КІНЕЦЬ ЯКЩО
        // 2. Чи не закінчився цикл? (неявна перевірка і < n)
    КІНЕЦЬ ДЛЯ
    ПОВЕРНУТИ -1
КІНЕЦЬ ФУНКЦІЇ
```

На кожній ітерації циклу виконуються дві операції порівняння:

1. Порівняння значення: `A[i] == key`.

2. Порівняння індексу: `i < n` (ця перевірка виконується на кожному кроці циклу `for`, щоб знати, чи потрібно його продовжувати).

На перший погляд, це дрібниця. Але у високопродуктивних системах, де такий пошук може виконуватися мільйони разів усередині іншого, більшого алгоритму, кожна зайва операція має значення. Виникає питання: чи можна позбутися однієї з цих перевірок?

1.2. Ідея "Вартового": гарантована знахідка

Відповідь — так, і для цього використовується елегантний трюк під назвою "пошук з бар'єром" або "пошук з вартовим".

Аналогія: Уявіть, що ви шукаєте свою машину на дуже довгому паркувальному ряду. Ви йдете вздовж нього і на кожному кроці думаєте: "Це моя машина? Ні. А чи не закінчився вже ряд?". Це дві перевірки. А тепер уявіть, що ви попросили друга (вартового) стати в самому кінці ряду. Тепер ваше завдання спрощується. Ви просто йдете і шукаєте свою машину, бо ви на 100% впевнені, що рано чи пізно ви когось знайдете — або свою машину, або друга. Вам більше не потрібно постійно перевіряти, чи не закінчився ряд.

Ідея алгоритму: Перед початком пошуку ми штучно додаємо шуканий елемент (`key`) в кінець масиву. Цей доданий елемент і є нашим "вартовим". Тепер ми можемо бути впевнені, що цикл пошуку гарантовано знайде потрібне значення і завершиться. Це дозволяє нам прибрати з тіла циклу перевірку на вихід за межі масиву, залишивши лише одне порівняння — порівняння значення.

Після того, як цикл завершився, нам залишається лише перевірити, де саме він зупинився.

– Якщо він зупинився на позиції, що менша за початковий розмір масиву, — вітаємо, ми знайшли справжній елемент.
– Якщо він зупинився на позиції нашого "вартового", — це означає, що справжнього елемента в масиві не було.

1.3. Псевдокод та його аналіз

Щоб не збільшувати розмір масиву, ми тимчасово замінимо останній елемент на наш "вартовий".

```
// Вхід: A - масив даних, key - значення, яке потрібно знайти.
// Вихід: індекс першого входження key в A, або -1, якщо key не знайд
ено.
```

```
ФУНКЦІЯ SentinelLinearSearch(Array A, key):
    n = довжина(A)

    // Якщо масив порожній, одразу повертаємо -1
    ЯКЩО n == 0 ТО ПОВЕРНУТИ -1 КІНЕЦЬ ЯКЩО

    // Крок 1: Зберігаємо останній елемент і ставимо на його місце "в
артового".
    last_element = A[n - 1]
    A[n - 1] = key

    // Крок 2: Запускаємо цикл, який тепер має лише ОДНУ перевірку.
    i = 0
    ПОКИ A[i] != key:
        i = i + 1
    КІНЕЦЬ ПОКИ

    // Крок 3: Повертаємо останній елемент на своє місце, щоб не змін
ити вхідний масив.
    A[n - 1] = last_element

    // Крок 4: Аналізуємо результат.
    // Якщо цикл зупинився раніше останнього індексу, АБО якщо останн
ій елемент
    // (який ми зберегли) і був тим, що ми шукали, то елемент знайден
о.
    ЯКЩО i < n - 1 АБО last_element == key ТО
        ПОВЕРНУТИ i
    ІНАКШЕ
        // Інакше ми дійшли до кінця і знайшли лише нашого "вартового
".
        ПОВЕРНУТИ -1
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Аналіз складності та реальна продуктивність

- Асимптотична складність: Давайте будемо чесними: з точки зору нотації Big O, нічого не змінилося. У найгіршому випадку ми все одно проходимо по всьому масиву, виконуючи n порівнянь. Часова складність: $O(n)$. Просторова складність: $O(1)$.
- Реальна продуктивність: То в чому ж сенс? Сенс у мікрооптимізації. Ми зменшили кількість операцій всередині циклу. У стандартному лінійному пошуку на кожній ітерації ми маємо дві операції

порівняння (`A[i] == key` та `i < n`). У версії з вартовим — лише одна (`A[i] != key`). Для сучасних процесорів це означає менше умовних переходів (branching), що дозволяє конвеєру процесора працювати ефективніше. На дуже великих масивах і в коді, де продуктивність є критичною (наприклад, у низькорівневих бібліотеках), ця, на перший погляд, незначна різниця може дати помітний приріст швидкості (5-20% залежно від архітектури).

2. Область застосування

Цей алгоритм є класичним прикладом низькорівневої оптимізації.

- Реалізація стандартних бібліотек: Подібні техніки використовуються в реалізаціях функцій пошуку в стандартних бібліотеках мов C/C++, де кожен такт процесора має значення.
- Вбудовані системи: У системах з обмеженими ресурсами, де потрібно “вичавити” максимум продуктивності з простого апаратного забезпечення.
- Внутрішні цикли складних алгоритмів: Коли лінійний пошук є невеликою, але дуже часто викликаною частиною іншого, більшого алгоритму.

3. Приклади застосування

- Пошук символу в рядку: Деякі реалізації функції **`strchr`** у мові C використовують схожий підхід для пошуку першого входження символу в рядок.
- Фільтрація даних у реальному часі: У системах, що обробляють потоки даних, де потрібно швидко перевіряти кожен новий елемент на відповідність умові, зменшення операцій у циклі може бути критичним.

Переваги та недоліки

Головна перевага пошуку з бар’єром — це його практична швидкість. Завдяки зменшенню кількості перевірок всередині циклу, він виконується швидше за класичний лінійний пошук, хоча і має таку ж асимптотичну складність $O(n)$.

Однак є і суттєві недоліки. По-перше, алгоритм модифікує вхідний масив, тимчасово перезаписуючи його останній елемент. Це може бути неприйнятним, якщо масив є спільним ресурсом у багатопотоковій програмі або якщо він має бути лише для читання. По-друге, реалізація є

трохи складнішою і вимагає додаткових кроків для збереження та відновлення останнього елемента, що може нівелювати виграш у швидкості на малих масивах.

Контрольні запитання

1. Яку саме неефективність стандартного лінійного пошуку усуває пошук з бар'єром?
2. Поясніть роль "вартового" в алгоритмі. Чому він гарантує завершення циклу?
3. Чому, незважаючи на зменшення кількості операцій, асимптотична складність $O(n)$ не змінюється?
4. Назвіть головний недолік пошуку з бар'єром. У якій ситуації його не можна застосовувати?
5. Чи дасть пошук з бар'єром значний приріст продуктивності на масиві з 10 елементів? Чому?

Методичні вказівки до лабораторної роботи

1. Назва

Емпіричне порівняння продуктивності стандартного лінійного пошуку та пошуку з бар'єром.

2. Мета

Практично переконатись у перевагах мікрооптимізації. Навчитись вимірювати час виконання алгоритмів та аналізувати, як невелика зміна в логіці впливає на реальну швидкість.

3. Завдання

Створити програму, яка реалізує дві функції: стандартний лінійний пошук та пошук з бар'єром. Провести серію експериментів для вимірювання часу їх роботи на великому масиві даних.

4. Опис

Програма має містити обидві реалізації. Для отримання статистично значущих результатів, кожний пошук потрібно буде виконати велику кількість разів (наприклад, 1,000,000 разів) на великому масиві.

5. Порядок виконання

1. Реалізуйте обидві функції: Напишіть функції `StandardLinearSearch` та `SentinelLinearSearch`.

2. Створіть великий масив: Згенеруйте масив розміром щонайменше 10,000 елементів. Заповніть його випадковими числами.
3. Підготуйте тест: Оберіть для пошуку елемент, якого гарантовано немає в масиві, щоб змусити обидва алгоритми пройти по всіх елементах (найгірший випадок).
4. Проведіть вимірювання:
    - Запустіть цикл, що виконує `StandardLinearSearch` 1,000,000 разів. Заміряйте загальний час виконання цього циклу.
    - Запустіть інший цикл, що виконує `SentinelLinearSearch` 1,000,000 разів. Заміряйте його час.
5. Порівняйте результати: Виведіть на екран час роботи для обох алгоритмів та порівняйте їх.

6. Результат

Програма, що демонструє, що пошук з бар'єром на практиці працює швидше за стандартний лінійний пошук на великій кількості ітерацій.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією обох функцій та коду для вимірювання часу.
2. Продемонструвати роботу додатку та показати отримані результати вимірювань.
3. Бути готовим відповісти на питання:
    - Пояснити, чому виникла різниця у часі виконання.
    - Як працює логіка обробки останнього елемента у вашій реалізації пошуку з бар'єром?
    - Чи буде різниця у продуктивності такою ж помітною, якщо масив буде дуже малим (напр., 5 елементів)?

Тести

1. Головна мета пошуку з бар'єром — це:

a) Покращити асимптотичну складність з $O(n)$ до $O(\log n)$.

b) Зменшити кількість порівнянь всередині циклу пошуку з двох до одного.

c) Дозволити пошук у невпорядкованих масивах.

d) Використовувати менше пам’яті.

2. Яка дія є першим кроком в алгоритмі пошуку з бар’єром?

a) Запуск циклу `while`.

b) Перевірка, чи не порожній масив.

c) Сортування масиву.

d) Збереження останнього елемента та заміна його на шуканий ключ.

3. Яка часова складність пошуку з бар’єром у найгіршому випадку?

a) $O(1)$

b) $O(\log n)$

c) $O(n)$

d) $O(n^2)$

4. Пошук з бар’єром НЕ можна застосовувати, якщо:

a) Масив не відсортований.

b) Масив містить дублікати.

c) Масив доступний лише для читання (read-only).

d) Масив дуже великий.

5. Після завершення циклу `while` в алгоритмі з бар’єром, як визначити, чи був знайдений справжній елемент?

a) Якщо індекс `i` менший за `n-1`, елемент точно знайдено.

b) Потрібно порівняти індекс `i` з `n-1` та додатково перевірити збережений останній елемент.

c) Якщо цикл завершився, елемент ніколи не знайдено.

d) Потрібно запустити двійковий пошук.

Відповіді: 1-b, 2-d, 3-c, 4-c, 5-b

Тести підвищеної складності

1. У якому з наступних сценаріїв використання пошуку з бар’єром, ймовірно, буде менш ефективним, ніж стандартний лінійний пошук, навіть на великій кількості ітерацій?

a) Коли масив містить переважно унікальні значення.

b) Коли пошук виконується у функції, що викликається з іншої функції, створюючи додаткові накладні витрати на стек.

c) Коли масив дуже малий (напр., 3-5 елементів), і вартість збереження/відновлення останнього елемента є сумірною з вартістю зекономлених перевірок.

d) Коли масив відсортований у зворотному порядку.

2. Розглянемо реалізацію пошуку з бар'єром. Якщо вихідний масив `A` вже містить шуканий ключ `key` на своїй останній позиції `n-1`, як коректний алгоритм має визначити результат?

a) Алгоритм негайно завершить цикл на першій ітерації (`i=0`) і поверне `n-1`.

b) Цикл `while (A[i] != key)` завершиться при `i = n-1`. Подальша перевірка `last_element == key` дасть `true`, і алгоритм коректно поверне `n-1`.

c) Алгоритм не зможе відрізнити справжній елемент від "вартового" і помилково поверне -1.

d) Алгоритм увійде в нескінченний цикл, оскільки "вартовий" збігається з існуючим елементом.

3. Спроба реалізувати пошук з бар'єром для однозв'язного списку (де ми не зберігаємо вказівник на хвіст) призведе до:

a) Значного прискорення, оскільки кількість порівнянь у циклі зменшиться.

b) Складності $O(\log n)$ через усунення перевірки на `NULL`.

c) Втрати всього виграшу в продуктивності, оскільки для встановлення "вартового" в кінець списку все одно потрібно пройти по ньому ($O(n)$), що нівелює оптимізацію циклу.

d) Неможливості реалізації, оскільки у вузла списку немає індексу.

4. Основна причина, чому пошук з бар'єром працює на практиці швидше за стандартний лінійний пошук, полягає у:

a) Зменшенні кількості операцій доступу до пам'яті.

b) Зменшенні кількості умовних переходів (branching) всередині циклу, що дозволяє конвеєру процесора працювати ефективніше.

c) Покращенні кеш-локальності через більш передбачуваний доступ до даних.

d) Зменшенні асимптотичної складності з $O(n)$ до $O(n/\log n)$.

5. Як потрібно модифікувати алгоритм пошуку з бар'єром для ефективного знаходження останнього входження елемента в масив?

a) Просто запустити цикл з кінця масиву до початку, встановивши "вартового" на нульовий елемент.

b) Алгоритм неможливо адаптувати для цієї задачі, потрібно використовувати стандартний лінійний пошук.

c) Потрібно спершу перевернути масив, знайти перше входження, а потім обчислити початковий індекс.

d) Потрібно встановити "вартового", знайти перше входження, запам'ятати його, і продовжити пошук до кінця, оновлюючи індекс кожного разу, коли елемент знайдено.

Відповіді

1. c) На дуже малих масивах фіксовані накладні витрати на збереження (`last_element = A[n-1]`), запис (`A[n-1] = key`), відновлення (`A[n-1] = last_element`) та фінальну перевірку (`if...`) можуть "з'їсти" весь виграш від усунення однієї перевірки в тілі короткого циклу.

2. b) Цикл зупиниться на останньому елементі. Подальша логіка `ЯКЩО i < n - 1 АБО last_element == key ТО` коректно обробить цей випадок, оскільки друга частина умови (`last_element == key`) буде істинною.

3. c) Щоб додати "вартового" в кінець однозв'язного списку, потрібно спочатку дійти до цього кінця, що само по собі є операцією $O(n)$. Це повністю нівелює будь-яку вигоду від оптимізації циклу, яка економить лише $O(n)$ операцій порівняння.

4. b) Кожен умовний оператор `if` або умова циклу `while`/`for` транслюється в інструкції умовного переходу для процесора. Такі переходи можуть порушувати роботу конвеєра (pipeline), змушуючи його скидати проміжні результати. Менше переходів — плавніша робота конвеєра і вища реальна продуктивність.

5. a) Найефективніша адаптація — це дзеркальне відображення алгоритму. Ми зберігаємо `A[0]`, ставимо на його місце "вартового", і запускаємо цикл з `i = n-1` до `0`. Це зберігає перевагу однієї перевірки в циклі.

Завдання для самостійної роботи

1. Пошук з кінця. Напишіть версію пошуку з бар'єром, яка шукає елемент з кінця масиву до початку.
2. Без збереження. Модифікуйте алгоритм так, щоб не зберігати останній елемент у тимчасову змінну. Натомість, спершу перевірте, чи не є останній елемент шуканим, а вже потім запускайте цикл по `n-1` елементах. Порівняйте кількість операцій.
3. Пошук у зв'язаному списку. Чи можна застосувати ідею "вартового" до однозв'язного списку? Опишіть, як би ви це реалізували і які переваги/недоліки це мало б.
4. Аналіз інструкцій. Дослідіть, на які інструкції процесора транслюється стандартний цикл `for` та цикл `while` у версії з бар'єром. Спробуйте пояснити різницю у швидкості на низькому рівні.
5. Пошук першого парного. Напишіть функцію пошуку з бар'єром, яка знаходить не конкретне значення, а перший елемент, що задовольняє умові (наприклад, є парним).
6. 2D-масив. Подумайте, як можна було б застосувати ідею "вартового" для пошуку елемента у двовимірній матриці.
7. Пошук кількох елементів. Вам дано масив `A` та масив ключів `keys`. Напишіть програму, що знаходить індекси всіх ключів у масиві `A`, використовуючи пошук з бар'єром для кожного ключа.
8. Об'єднання з двійковим пошуком. Чи має сенс об'єднувати ідею "вартового" з двійковим пошуком? Обґрунтуйте свою відповідь.
9. Есе-аналіз. Напишіть есе на тему "Мікрооптимізації: коли вони є виправданими, а коли — передчасними?". Використовуйте пошук з бар'єром як основний приклад.

   Реалізація для рядків. Напишіть функцію `Sentinel_strchr`, яка знаходить перше входження символу в рядок (масив символів, що закінчується `\0`) за допомогою техніки "вартового".

Інші методи пошуку

Пошук стрибками

Ми вже знаємо два крайні випадки: повільний, але універсальний лінійний пошук, та надзвичайно швидкий, але вимогливий до сортування двійковий пошук. А що, якщо існує щось середнє? Алгоритм, який швидший за

лінійний, але, можливо, простіший або цікавіший за двійковий. Такий алгоритм існує, і це — пошук стрибками.

1. Теоретичні відомості

1.1. Ідея пошуку стрибками: компроміс між послідовністю та діленням

Після знайомства з лінійним ($O(n)$) та двійковим ($O(\log n)$) пошуками виникає питання: чи можна знайти компроміс між послідовним переглядом кожного елемента та постійним діленням масиву навпіл?

Уявіть, що ви знову шукаєте потрібну сторінку у великій книзі. Лінійний пошук — це гортати по одній сторінці. Двійковий — відкривати точно посередині. А пошук стрибками — це швидко прогортати книгу фіксованими "пачками" сторінок, наприклад, по 20 за раз. Ви гортаєте: 20-та, 40-ва, 60-та, 80-та… Якщо потрібна вам сторінка — 75-та, ви зупинитесь на 80-й, зрозумівши, що "перестрибнули" її. Тепер ви знаєте, що ваша сторінка знаходиться десь у попередньому блоці, між 60-ю та 80-ю сторінками. І вже в цьому маленькому блоці ви можете знайти її звичайним лінійним переглядом.

Це і є пошук стрибками. Його ідея полягає у наступному:

1. Масив (обов'язково відсортований) розбивається на блоки фіксованого розміру.
2. Алгоритм "стрибає" по кінцях цих блоків, доки не знайде блок, в якому може знаходитись шуканий елемент.
3. Після знаходження такого блоку, в ньому виконується звичайний лінійний пошук.

Цей підхід дозволяє швидко пропустити великі частини масиву, які гарантовано не містять потрібного елемента, але не вимагає складних обчислень середини, як у двійковому пошуку.

1.2. Визначення оптимального розміру стрибка

Продуктивність алгоритму напряму залежить від розміру блоку (або кроку стрибка), який ми оберемо. Позначимо розмір масиву як n, а розмір блоку — як m.

– У найгіршому випадку нам доведеться зробити n/m стрибків, щоб знайти правильний блок.
– Після цього нам доведеться виконати до $m - 1$ порівнянь всередині цього блоку за допомогою лінійного пошуку.

Загальна кількість порівнянь у найгіршому випадку приблизно дорівнює: $f(m) = \frac{n}{m} + m - 1$. Нам потрібно знайти такий розмір блоку m, при якому ця функція буде мінімальною. З математичного аналізу відомо, що функція досягає мінімуму, коли її похідна дорівнює нулю. Знайшовши похідну $f(m)$ по m та прирівнявши її до нуля, ми отримаємо, що оптимальне значення $m = \sqrt{n}$.

Таким чином, оптимальний розмір стрибка для пошуку стрибками дорівнює квадратному кореню з розміру масиву. Це інтуїтивно зрозуміло: ми балансуємо вартість першої фази (стрибки) та другої (лінійний пошук).

1.3. Псевдокод та його аналіз

```
// Вхід: A - відсортований масив, key - значення, яке потрібно знайти
.
// Вихід: індекс знайденого елемента, або -1, якщо не знайдено.
ФУНКЦІЯ JumpSearch(Array A, key):
    n = довжина(A)

    // Крок 1: Визначаємо розмір стрибка.
    step = ціла_частина(sqrt(n))

    // Крок 2: Крок  стрибків.
    // Знаходимо блок, де може знаходитись елемент.
    prev = 0
    ПОКИ A[min(step, n) - 1] < key:
        prev = step
        step = step + ціла_частина(sqrt(n))
        // Якщо ми вийшли за межі масиву, елемента немає.
        ЯКЩО prev >= n ТО
            ПОВЕРНУТИ -1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ

    // Крок 3: Крок  лінійного пошуку.
    // Виконуємо лінійний пошук у знайденому блоці [prev ... step].
    ПОКИ A[prev] < key:
        prev = prev + 1
        // Якщо ми вийшли за межі блоку або масиву, елемента немає.
        ЯКЩО prev == min(step, n) ТО
            ПОВЕРНУТИ -1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ

    // Крок 4: Перевіряємо, чи знайдено елемент.
```

```
    ЯКЩО A[prev] == key ТО
        ПОВЕРНУТИ prev
    КІНЕЦЬ ЯКЩО

    ПОВЕРНУТИ -1 // Елемент не знайдено
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Аналіз складності

- Часова складність: При оптимальному розмірі кроку $m = \sqrt{n}$, кількість операцій у найгіршому випадку складається з $\approx \sqrt{n}$ стрибків та $\approx \sqrt{n}$ кроків лінійного пошуку. Загальна складність: $O(\sqrt{n})$. Це значно краще, ніж $O(n)$ у лінійного пошуку, але гірше, ніж $O(\log n)$ у двійкового.
- Просторова складність: Алгоритм використовує лише декілька змінних для зберігання поточних позицій (`step`, `prev`), тому його просторова складність є константною. Просторова складність: $O(1)$.

2. Область застосування

Пошук стрибками є цікавим теоретичним алгоритмом, що демонструє компроміс між двома крайностями. На практиці, на сучасних комп'ютерах з швидкою оперативною пам'яттю, він майже завжди буде повільнішим за двійковий пошук.

Однак, він може мати перевагу в дуже специфічних умовах, наприклад, в системах, де вартість "стрибка назад" є значно вищою, ніж рух уперед. У двійковому пошуку ми постійно стрибаємо то вліво, то вправо по масиву. У пошуку стрибками ми робимо лише один стрибок назад (імпліцитно, коли починаємо лінійний пошук з `prev`), а решта рухів — уперед.

3. Приклади застосування

- Педагогічна мета: Найчастіше використовується в навчальних курсах для ілюстрації того, що між лінійним та логарифмічним пошуком існують проміжні варіанти, та для пояснення ідеї оптимізації через вибір розміру блоку.
- Теоретична перевага: На деяких застарілих типах пам'яті, як-от магнітні стрічки, де перемотка назад була дуже повільною, такий підхід міг би бути виправданим.

Переваги та недоліки

Головною перевагою пошуку стрибками є те, що він значно ефективніший за лінійний пошук. Алгоритм простий для розуміння і не вимагає рекурсії,

що робить його дещо легшим в реалізації, ніж рекурсивна версія двійкового пошуку. Також цікавим є той факт, що в середньому пошук стрибками торкається меншої кількості елементів пам'яті, ніж двійковий пошук, оскільки він "перестрибує" через більшість даних.

З іншого боку, його основний недолік — це асимптотична складність $O(\sqrt{n})$, яка є гіршою за $O(\log n)$ у двійкового пошуку. Для великих наборів даних ця різниця стає колосальною. Як і двійковий пошук, він має строгу вимогу до впорядкованості даних, що обмежує його застосування. На сучасних комп'ютерних архітектурах його практичні переваги майже нівелюються, і двійковий пошук залишається беззаперечним лідером для пошуку у відсортованих масивах.

Контрольні запитання

4. У чому полягає основна ідея пошуку стрибками?
5. Чому оптимальний розмір стрибка дорівнює $\sqrt{n}$? Поясніть компроміс, який лежить в основі цього вибору.
6. Порівняйте часову складність лінійного, двійкового пошуку та пошуку стрибками.
7. Яка головна вимога до даних для застосування пошуку стрибками?
8. Наведіть приклад гіпотетичної системи, де пошук стрибками міг би бути ефективнішим за двійковий.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму пошуку стрибками.

2. Мета

Практичне засвоєння логіки алгоритму пошуку стрибками та аналіз його поведінки на практиці.

3. Завдання

Створити програму, яка реалізує функцію пошуку стрибками та демонструє її роботу на відсортованому масиві цілих чисел.

4. Опис

Програма повинна містити функцію `JumpSearch`, яка приймає масив та ключ. Головна частина програми повинна створювати тестовий масив, сортувати його, а потім викликати функцію для пошуку різних елементів.

5. Порядок виконання

1. Реалізуйте функцію: Напишіть функцію `JumpSearch`, керуючись наведеним псевдокодом.
2. Створіть тестовий масив: В основній частині програми створіть масив з 20-30 цілих чисел та відсортуйте його.
3. Обчисліть крок: Обчисліть оптимальний розмір кроку для вашого масиву.
4. Реалізуйте фазу стрибків: Напишіть цикл `while`, що переміщується по блоках. Додайте виведення на екран, щоб бачити, через які індекси "стрибає" алгоритм.
5. Реалізуйте фазу лінійного пошуку: Після знаходження потрібного блоку, запустіть лінійний пошук. Також додайте виведення, щоб бачити, які індекси перевіряються.
6. Протестуйте: Перевірте роботу для елемента на початку, в середині, в кінці масиву, а також для елемента, що відсутній.

6. Результат

Додаток, що коректно знаходить елементи у відсортованому масиві за допомогою пошуку стрибками та наочно демонструє етапи своєї роботи.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією функції `JumpSearch`.
2. Продемонструвати роботу додатку, показавши трасування алгоритму (виведення індексів, що перевіряються) для різних сценаріїв.
3. Бути готовим відповісти на питання:
    - Пояснити, як ви обчислюєте розмір стрибка.
    - Як ваш код обробляє випадок, коли шуканий елемент більший за всі елементи масиву?
    - Порівняйте кількість порівнянь, яку робить ваш алгоритм, з кількістю порівнянь для лінійного та двійкового пошуку для того ж масиву.

Тести

1. Яка часова складність алгоритму пошуку стрибками?

a) $O(\log n)$

b) $O(n)$

c) $O(n\log n)$

d) $O(\sqrt{n})$

2. Оптимальний розмір блоку для пошуку стрибками в масиві з N елементів дорівнює:

a) $N/2$

b) $\log_2 N$

c) $\sqrt{N}$

d) 10

3. Пошук стрибками є компромісом між:

a) Сортуванням бульбашкою та швидким сортуванням.

b) Лінійним пошуком та двійковим пошуком.

c) Масивом та зв'язним списком.

d) Рекурсією та ітерацією.

4. Яка Крок слідує одразу за фазою "стрибків" в алгоритмі?

a) Двійковий пошук у знайденому блоці.

b) Повторні стрибки з меншим кроком.

c) Лінійний пошук у знайденому блоці.

d) Завершення алгоритму.

5. Для якої структури даних пошук стрибками НЕ має сенсу?

a) Відсортований масив.

b) Відсортований двозв'язний список.

c) Невпорядкований масив.

d) Для всіх перелічених він має сенс.

Відповіді: 1-d, 2-c, 3-b, 4-c, 5-c

Тести підвищеної складності

1. Основна мета при виборі оптимального розміру кроку `m = sqrt(n)` для пошуку стрибками полягає у:

a) Мінімізації кількості стрибків, ігноруючи вартість лінійного пошуку.

b) Мінімізації загальної кількості порівнянь шляхом балансування вартості фази стрибків (n/m) та вартості фази лінійного пошуку ($m - 1$).

c) Гарантуванні, що кількість порівнянь ніколи не перевищить $n/2$.

d) Адаптації алгоритму для роботи з невпорядкованими даними.

2. У відсортованому масиві з 1,000,000 елементів, яка з наступних ситуацій призведе до приблизно однакової кількості порівнянь?

a) Лінійний пошук елемента в середині та двійковий пошук.

b) Пошук стрибками та лінійний пошук елемента на позиції $\sqrt{n}$.

c) Двійковий пошук та пошук стрибками.

d) Пошук стрибками та пошук елемента, що знаходиться на позиції 10.

3. Якщо спробувати реалізувати пошук стрибками на двозв'язному списку (який відсортований, але не має доступу за індексом $O(1)$), якою буде його фінальна часова складність?

a) $O(\sqrt{n})$, оскільки двозв'язний список дозволяє рухатись в обидва боки.

b) $O(\log n)$, оскільки стрибки прискорюють пошук.

c) $O(n)$, оскільки кожен "стрибок" на `m` позицій вперед все одно вимагатиме `m` послідовних переходів по вказівниках.

d) $O(1)$, якщо список є циклічним.

4. Для масиву з 99 елементів та оптимального кроку стрибка, пошук якого з наступних елементів (за індексом) вимагатиме максимальної кількості порівнянь у найгіршому випадку?

a) Елемента з індексом 98.

b) Елемента з індексом 8.

c) Елемента з індексом 89.

d) Елемента з індексом 49.

5. Що станеться, якщо помилково застосувати алгоритм пошуку стрибками до невпорядкованого масиву?

a) Алгоритм завжди буде працювати коректно, але повільніше.

b) Алгоритм може повернути неправильний результат (наприклад, `-1`, хоча елемент присутній), оскільки він пропустить потрібний блок.

c) Алгоритм увійде в нескінченний цикл.

d) Алгоритм автоматично відсортує масив перед пошуком.

Відповіді

1. b) Оптимізація полягає саме в тому, щоб знайти такий розмір кроку `m`, при якому сума операцій обох фаз ($n/m + m$) буде мінімальною. Це досягається, коли обидва доданки приблизно рівні, що і призводить до $m = \sqrt{n}$.

2. b) Пошук стрибками має складність $O(\sqrt{n})$. Лінійний пошук елемента на позиції $\sqrt{n}$ також вимагатиме $\sqrt{n}$ порівнянь. Інші пари мають кардинально різну складність.

3. c) Перевага стрибків базується на можливості доступу до індексу $i + m$ за $O(1)$. У зв'язному списку, щоб "стрибнути" на `m` позицій, потрібно пройти `m` вузлів. В результаті перша Крок алгоритму виродиться в лінійний прохід, і загальна складність буде $O(n)$.

4. a) Крок стрибка для 99 елементів буде $\sqrt{99} \approx 9$. Стрибки будуть по індексах 9, 18, 27, ..., 81, 90, 99 (кінець). Щоб знайти елемент на індексі 98, потрібно зробити 10 стрибків (до кінця) і потім до 9 кроків лінійного пошуку назад (з 98 до 90). Це дасть максимальну кількість операцій (близько 19). Для індекса 89 знадобиться 10 стрибків і 0 кроків назад.

5. b) Алгоритм базується на припущенні, що якщо `A[i] < key`, то всі елементи до `A[i]` також менші за `key`. Якщо масив не відсортований, це припущення хибне. Алгоритм може "перестрибнути" через шуканий елемент, а потім шукати його в неправильному блоці, в результаті чого не знайде його.

Завдання для самостійної роботи

1. Аналіз кроку. Проведіть емпіричний аналіз: як змінюється кількість порівнянь в алгоритмі пошуку стрибками, якщо розмір кроку дорівнює не $\sqrt{n}$, а, наприклад, 20 (константа) або $\log n$?
2. Пошук назад. Модифікуйте псевдокод так, щоб лінійний пошук на другому етапі йшов не вперед (від `prev`), а назад (від `step`).
3. Перше входження. Напишіть версію пошуку стрибками, яка гарантовано знаходить індекс першого входження елемента, якщо в масиві є дублікати.
4. Останнє входження. Аналогічно, напишіть функцію для пошуку останнього входження.

5. Двозв’язний список. Чи можна застосувати пошук стрибками до відсортованого двозв’язного списку? Якщо так, яка буде його складність? Реалізуйте такий пошук.
6. Тернарний пошук. Дослідіть алгоритм тернарного пошуку. Порівняйте його ідею, складність та область застосування з пошуком стрибками та двійковим пошуком.
7. Адаптивний крок. Подумайте над ідеєю адаптивного пошуку стрибками, де розмір наступного стрибка залежить від того, наскільки далеко від шуканого значення знаходиться поточний елемент. Опишіть можливий алгоритм.
8. Теоретичне порівняння. Напишіть доведення, чому функція $f(m) = n/m + m - 1$ досягає мінімуму при $m = \sqrt{n}$.
9. Кількість порівнянь. Виведіть точну формулу для максимальної кількості порівнянь, яку виконує пошук стрибками для масиву з n елементів та кроком $\sqrt{n}$.
10. Есе-аналіз. Напишіть есе на тему: “Чому, незважаючи на існування двійкового пошуку, вивчення таких алгоритмів, як пошук стрибками, є корисним для розвитку алгоритмічного мислення?”.

Інтерполяційний пошук

Ми вже знаємо, що двійковий пошук, який завжди перевіряє середину, є значно ефективнішим за лінійний. Але чи можна зробити ще “розумніший” вибір? Чи можна не “сліпо” ділити масив навпіл, а робити припущення про те, де, ймовірно, знаходиться елемент? Саме цю ідею реалізує інтерполяційний пошук.

1. Теоретичні відомості

1.1. Ідея інтерполяційного пошуку: розумний стрибок

Згадаймо нашу аналогію з паперовим словником. Двійковий пошук — це відкрити книгу рівно посередині. Але якщо ви шукаєте слово “Яблуко”, ваша інтуїція підказує, що потрібно шукати в самому кінці алфавіту, а не в середині на літеру “М”. Ви подумки інтерполюєте — робите припущення про позицію слова, базуючись на його алфавітному порядку.

Інтерполяційний пошук — це вдосконалення двійкового пошуку, яке використовує цю інтуїцію. Замість того, щоб завжди перевіряти середній елемент, він обчислює ймовірну позицію шуканого елемента, враховуючи значення першого та останнього елементів у поточному діапазоні пошуку.

Ключова ідея: Якщо шукане значення `key` близьке до початкового елемента масиву `A[left]`, то його ймовірна позиція буде знаходитись ближче до індексу `left`. Якщо ж `key` близький до кінцевого елемента `A[right]`, то його позиція, скоріше за все, буде ближче до `right`.

Цей підхід працює, але, як і двійковий пошук, він має сувору вимогу: масив має бути відсортованим. Більше того, для своєї максимальної ефективності він передбачає, що дані в масиві розподілені рівномірно.

1.2. Формула для обчислення позиції

Як саме алгоритм робить свій "освічений здогад"? Він використовує формулу лінійної інтерполяції. Уявіть, що ми проводимо пряму лінію через дві точки: `(left, A[left])` та `(right, A[right])`, де перша координата — індекс, а друга — значення. Ми хочемо знайти на цій лінії таку точку (індекс `pos`), якій відповідає наше значення `key`.

Формула для обчислення цієї позиції виглядає так:

$$pos = \text{left} + \frac{(\text{key} - A[\text{left}]) \cdot (\text{right} - \text{left})}{A[\text{right}] - A[\text{left}]}$$

Давайте детально розберемо кожен компонент цієї формули:

- `key - A[left]`: Наскільки далеко наше шукане значення від значення на лівій межі.
- `A[right] - A[left]`: Загальний діапазон значень у поточному відрізку пошуку.
- Дріб $\frac{(\text{key}-A[\text{left}])}{A[\text{right}]-A[\text{left}]}$: Це найважливіша частина. Вона обчислює, яку пропорцію від загального діапазону значень складає відстань від нашого ключа до початкового значення. Наприклад, якщо ця пропорція дорівнює 0.8, це означає, що наш ключ знаходиться приблизно на 80% шляху від `A[left]` до `A[right]`.
- `(right - left)`: Загальний діапазон індексів.
- `left + ...`: Ми множимо знайдену пропорцію на діапазон індексів і додаємо до початкового індексу `left`, щоб знайти ймовірний індекс нашого елемента.

Приклад: Шукаємо число 80 у масиві `{10, 20, ..., 100}` (10 елементів, індекси від 0 до 9).

- `left = 0`, `A[left] = 10`

- `right = 9`, `A[right] = 100`
- `key = 80` `pos = 0 + ((80 - 10) * (9 - 0)) / (100 - 10) = 70 * 9 / 90 = 7`. Алгоритм одразу перевірить 7-й індекс, де і знаходиться число 80, знайшовши його за один крок! Двійковий пошук почав би з 4-го індексу.

1.3. Псевдокод та його аналіз

Реалізація інтерполяційного пошуку дуже схожа на ітеративну версію двійкового, але з іншою формулою для обчислення позиції.

```
// Вхід: A - відсортований масив, key - значення, яке потрібно знайти
.
// Вихід: індекс знайденого елемента, або -1, якщо не знайдено.
ФУНКЦІЯ InterpolationSearch(Array A, key):
    left = 0
    right = довжина(A) - 1

    // Цикл триває, доки межі не перетнулись і ключ знаходиться в діа
пазоні [A[left], A[right]]
    ПОКИ left <= right І key >= A[left] І key <= A[right]:
        // Якщо left == right, ми або знайшли елемент, або ні
        ЯКЩО left == right ТО
            ЯКЩО A[left] == key ТО ПОВЕРНУТИ left ІНАКШЕ ПОВЕРНУТИ -1
КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ЯКЩО

        // Крок 1: Обчислюємо ймовірну позицію за формулою інтерполяц
ії
        pos = left + ціла_частина(((key - A[left]) * (right - left))
/ (A[right] - A[left]))

        // Крок 2: Перевіряємо знайдену позицію
        ЯКЩО A[pos] == key ТО
            ПОВЕРНУТИ pos
        КІНЕЦЬ ЯКЩО

        // Крок 3: Звужуємо діапазон пошуку
        ЯКЩО A[pos] < key ТО
            left = pos + 1
        ІНАКШЕ
            right = pos - 1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ
```

```
    ПОВЕРНУТИ -1
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Аналіз складності: від логарифмів до подвійних логарифмів

- Часова складність (середній випадок): Якщо дані в масиві розподілені рівномірно (наприклад, `{1, 2, 3, ...}` або `{10, 20, 30, ...}`), інтерполяційний пошук демонструє свою справжню силу. Простір пошуку скорочується значно швидше, ніж удвічі на кожному кроці. Асимптотична складність у цьому випадку становить $O(\log(\log n))$. Це значно швидше, ніж $O(\log n)$ у двійкового пошуку. Для масиву з мільярда елементів двійковий пошук зробить близько 30 порівнянь, а інтерполяційний — лише близько 5 ($\log_2(\log_2 10^9) \approx 4.9$).
- Часова складність (найгірший випадок): Це "ахіллесова п'ята" алгоритму. Якщо дані розподілені нерівномірно (наприклад, експоненційно: `{2, 4, 8, 16, ..., 2^n}`), то формула інтерполяції буде постійно робити погані припущення, перевіряючи позиції, дуже близькі до однієї з меж. У цьому випадку продуктивність може катастрофічно впасти до $O(n)$, що дорівнює складності лінійного пошуку.
- Просторова складність: Оскільки алгоритм є ітеративним і використовує лише кілька змінних, його просторова складність є константною: $O(1)$.

2. Область застосування

Інтерполяційний пошук є вузькоспеціалізованим алгоритмом, який показує найкращі результати за певних умов.

- Великі, рівномірно розподілені набори даних: Це ідеальний сценарій.
- Системи, де вартість доступу до пам'яті висока: Якщо кожне порівняння є дорогим (наприклад, дані зберігаються на повільному носії), то мінімізація кількості порівнянь (з 30 до 5, як у нашому прикладі) може дати суттєвий виграш.
- Статичні або рідко змінювані дані: Як і двійковий пошук, інтерполяційний вимагає відсортованого масиву, тому він не підходить для даних, що часто оновлюються.

3. Приклади застосування

– Пошук у телефонному довіднику: Імена людей в цілому розподілені за алфавітом досить рівномірно, що робить інтерполяційний пошук хорошим кандидатом.
– Словники: Слова у мові також розподілені відносно рівномірно.
– Пошук у великих таблицях бази даних по проіндексованому числовому полю, якщо значення в цьому полі розподілені рівномірно (наприклад, ID користувачів, що реєструються з постійною швидкістю).

Переваги та недоліки

Переваги інтерполяційного пошуку полягають у його феноменальній середній швидкості $O(\log(\log n))$ для підходящих наборів даних. Для дуже великих, рівномірно розподілених масивів він значно перевершує двійковий пошук за кількістю порівнянь.

Однак його недоліки є суттєвими. Головний з них — чутливість до розподілу даних. На нерівномірних даних, наприклад, якщо значення згруповані на початку масиву, а в кінці дуже розріджені, продуктивність алгоритму деградує до лінійного часу $O(n)$, що робить його абсолютно непридатним. Крім того, формула обчислення позиції складніша, ніж у двійковому пошуку, і вимагає більше арифметичних операцій, що на малих масивах може нівелювати перевагу від меншої кількості порівнянь.

Контрольні запитання

1. У чому полягає основна відмінність в ідеї інтерполяційного та двійкового пошуку?
2. Назвіть дві ключові вимоги до даних для ефективної роботи інтерполяційного пошуку.
3. Поясніть своїми словами, що представляє кожна частина формули для обчислення `pos`.
4. Опишіть сценарій (тип даних), при якому інтерполяційний пошук буде працювати зі складністю $O(n)$.
5. Чому, незважаючи на кращу середню складність, двійковий пошук на практиці використовується значно частіше, ніж інтерполяційний?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація інтерполяційного пошуку та порівняння його ефективності з двійковим пошуком.

2. Мета

Практичне засвоєння логіки алгоритму інтерполяційного пошуку. Дослідження його поведінки на різних наборах даних та порівняння продуктивності з двійковим пошуком.

3. Завдання

Створити програму, яка реалізує функцію інтерполяційного пошуку. Провести серію експериментів, порівнюючи кількість порівнянь, виконаних інтерполяційним та двійковим пошуком на рівномірно та нерівномірно розподілених даних.

4. Опис

Програма повинна містити реалізації обох алгоритмів. Основна частина програми має генерувати два типи тестових масивів та викликати обидві функції для пошуку елементів, підраховуючи кількість виконаних порівнянь.

5. Порядок виконання

1. Реалізуйте алгоритми: Напишіть функцію `InterpolationSearch` та функцію `BinarySearch` з попередніх розділів. Модифікуйте їх так, щоб вони повертали не тільки індекс, але й кількість виконаних порівнянь.
2. Створіть генератор даних:
    - Напишіть функцію, що генерує рівномірно розподілений відсортований масив (напр., `{10, 20, 30, ...}`).
    - Напишіть функцію, що генерує нерівномірно розподілений відсортований масив (напр., `{1, 2, 4, 8, 16, ...}`).
3. Проведіть експеримент:
    - Для обох типів масивів (напр., розміром 1000 елементів) виконайте пошук кількох елементів (на початку, в середині, в кінці) за допомогою обох алгоритмів.
    - Запишіть кількість порівнянь для кожного випадку в таблицю.
4. Проаналізуйте результати: Порівняйте отримані дані.

6. Результат

Програма, що демонструє роботу обох алгоритмів, та таблиця з результатами, що підтверджує теоретичні оцінки їхньої ефективності на різних типах даних.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією обох алгоритмів та підрахунком порівнянь.
2. Таблицю з результатами експериментів та короткі висновки.
3. Бути готовим відповісти на питання:
    - Пояснити, як працює формула інтерполяції у вашому коді.
    - На якому наборі даних інтерполяційний пошук виявився значно ефективнішим і чому?
    - На якому наборі даних його продуктивність деградувала і чому?

Тести

1. Інтерполяційний пошук є покращенням…

a) Лінійного пошуку

b) Пошуку стрибками

c) Двійкового пошуку

d) Хеш-таблиці

2. Яка часова складність інтерполяційного пошуку в середньому випадку для рівномірно розподілених даних?

a) $O(\log n)$

b) $O(\sqrt{n})$

c) $O(n)$

d) $O(\log(\log n))$

3. Для якого набору даних інтерполяційний пошук покаже найгіршу продуктивність?

a) `{10, 20, 30, 40, 50, 60}`

b) `{1, 10, 21, 32, 41, 53}`

c) `{1, 100, 101, 102, 103, 1000}`

d) `{1, 5, 10, 15, 20, 25}`

4. Що НЕ є обов’язковою вимогою для ефективної роботи інтерполяційного пошуку?

a) Масив має бути відсортованим.

b) Дані мають бути рівномірно розподілені.

c) Розмір масиву має бути степенем двійки.

d) Всі перелічені є обов'язковими.

5. Основна ідея інтерполяційного пошуку полягає у:

a) Діленні масиву на три частини.

b) Перевірці випадкового елемента.

c) Послідовному перегляді блоків.

d) Обчисленні ймовірної позиції елемента на основі його значення.

Відповіді: 1-c, 2-d, 3-c (дані сильно скупчені на початку та в кінці), 4-c, 5-d

Тести підвищеної складності

1. Який з наступних наборів даних, відсортованих за зростанням, з найбільшою ймовірністю призведе до деградації продуктивності інтерполяційного пошуку до $O(n)$?

a) Набір даних, де значення розподілені рівномірно, але з невеликим випадковим "шумом".

b) Набір даних з великою кількістю дублікатів у середині масиву.

c) Набір даних, де значення зростають експоненційно (напр., `{1, 2, 4, 8, 16, 32, ...}`).

d) Набір даних, де відсутній шуканий елемент.

2. При реалізації інтерполяційного пошуку, в якій ситуації може виникнути помилка ділення на нуль у формулі `pos = left + ((key - A[left]) * (right - left)) / (A[right] - A[left])`?

a) Коли шуканий ключ `key` дорівнює `A[left]`.

b) Коли `left` та `right` вказують на сусідні елементи.

c) Коли всі елементи в поточному діапазоні пошуку (`A[left]`...`A[right]`) мають однакове значення.

d) Коли масив містить від'ємні числа.

3. Ви працюєте з дуже великим відсортованим масивом, який загалом розподілений рівномірно, але має невеликі "кластери" з нерівномірним розподілом. Яка гібридна стратегія пошуку буде найбільш робастною та ефективною?

a) Завжди використовувати двійковий пошук, оскільки він має кращу гарантовану складність у найгіршому випадку.

b) Починати з інтерполяційного пошуку, але якщо розмір діапазону пошуку скорочується занадто повільно (напр., менше ніж вдвічі за 2-3 кроки), перемикатися на двійковий пошук.

c) Застосувати пошук стрибками, оскільки він менш чутливий до розподілу даних.

d) Використовувати інтерполяційний пошук для першої половини масиву, і двійковий — для другої.

4. Теоретичне обґрунтування середньої складності $O(\log(\log n))$ для інтерполяційного пошуку базується на припущенні, що:

a) На кожному кроці кількість порівнянь зменшується вдвічі.

b) Розмір простору пошуку на кожному кроці зменшується в середньому з `N` до $\sqrt{N}$.

c) Алгоритм використовує паралельні обчислення для перевірки кількох позицій одночасно.

d) На кожному кроці кількість біт, необхідних для представлення розміру діапазону, зменшується на константу.

5. Ідею інтерполяції для “розумного” звуження діапазону можна застосувати не тільки до масивів. Яка з наступних задач є найбільш близьким аналогом інтерполяційного пошуку?

a) Пошук найкоротшого шляху в графі за допомогою алгоритму A*, де евристика “здогадується” про відстань до цілі.

b) Метод хорд (False Position Method) для знаходження кореня неперервної функції на відрізку.

c) Сортування масиву методом швидкого сортування, де вибір опорного елемента є ключовим.

d) Пошук з поверненням, де ми відсікаємо безперспективні гілки.

Відповіді

1. c) В експоненційно зростаючому наборі даних переважна більшість значень буде сконцентрована в кінці діапазону. Формула інтерполяції, яка припускає лінійний розподіл, буде постійно робити дуже неточні припущення, перевіряючи позиції близько до `right` і скорочуючи діапазон лише на кілька елементів за крок, що вироджує пошук у лінійний.

2. c) Якщо `A[right] == A[left]`, знаменник (`A[right] - A[left]`) стає нулем. Це відбувається, коли всі елементи у поточному діапазоні однакові. Коректна реалізація повинна окремо обробляти цей випадок перед обчисленням `pos`.

3. b) Ця стратегія поєднує сильні сторони обох підходів. Вона використовує видатну середню швидкість інтерполяційного пошуку, але має "запобіжник": якщо він починає працювати погано через локальну нерівномірність даних, система перемикається на гарантовано стабільний двійковий пошук.

4. b) Це більш точне інтуїтивне пояснення складності $O(\log(\log n))$. Якщо двійковий пошук зменшує `N` до `N/2` (логарифмічне скорочення), то інтерполяційний пошук для ідеальних даних зменшує `N` приблизно до $\sqrt{N}$. Послідовне взяття квадратного кореня з числа значно швидше приводить до одиниці, ніж послідовне ділення навпіл.

5. b) Метод хорд працює за схожим принципом: замість того, щоб ділити відрізок навпіл (як метод бісекції, аналог двійкового пошуку), він проводить пряму лінію (хорду) через кінці відрізка і знаходить точку перетину цієї прямої з віссю X, використовуючи її як наступне, більш "розумне" наближення до кореня. Це прямий аналог інтерполяції.

Завдання для самостійної роботи

1. Рекурсивна реалізація. Напишіть рекурсивну версію інтерполяційного пошуку.
2. Доведення формули. Виходячи з рівняння прямої, що проходить через дві точки `(x1, y1)` та `(x2, y2)`, виведіть формулу для обчислення позиції в інтерполяційному пошуку.
3. Генератор найгіршого випадку. Напишіть програму, що генерує масив, на якому інтерполяційний пошук зробить максимальну кількість порівнянь.
4. Аналіз просторової складності. Оцініть та порівняйте просторову складність ітеративної та рекурсивної версій інтерполяційного пошуку.
5. Робота з дублікатами. Дослідіть, як поводитиме себе інтерполяційний пошук на відсортованому масиві з великою кількістю дублікатів. Чи потрібні модифікації?

6. Інтерполяція для рядків. Подумайте, чи можна застосувати інтерполяційний пошук для відсортованого масиву рядків. Як би ви визначили “відстань” між рядками для формули?
7. Гібридний пошук. Реалізуйте гібридний алгоритм, який починає з інтерполяційного пошуку, але якщо діапазон пошуку скорочується занадто повільно, він перемикається на двійковий пошук.
8. Обробка виходу за межі. Поясніть, чому умова `key >= A[left] I key <= A[right]` у циклі є важливою для коректності роботи алгоритму. Наведіть приклад, коли без неї виникне помилка.
9. Емпіричний аналіз. Проведіть експеримент для знаходження “точки перетину”, тобто розміру масиву, після якого інтерполяційний пошук на рівномірних даних стає стабільно швидшим за двійковий на вашому комп’ютері.
10. Пошук найближчого елемента. Модифікуйте алгоритм так, щоб він знаходив індекс елемента, найближчого до заданого `key`, якщо самого `key` в масиві немає.
11. Пошук першого входження. Адаптуйте алгоритм для пошуку першого входження елемента в масиві з дублікатами.
12. Порівняння з пошуком стрибками. Напишіть теоретичне порівняння інтерполяційного пошуку та пошуку стрибками. Які їхні спільні риси та ключові відмінності?
13. Потрійний пошук (Ternary Search). Дослідіть цей алгоритм. Для якого типу функцій він застосовується? Чи можна його застосувати для пошуку в масиві?
14. Дрібні числа. Як поводитиметься формула інтерполяції, якщо всі значення в масиві будуть дрібними (напр., від 0 до 1)? Чи потрібні зміни?
15. Есе-аналіз. Напишіть есе на тему “Чому інтерполяційний пошук, незважаючи на видатну середню складність, не став”вбивцею” двійкового пошуку в загальній практиці програмування?“.

Експоненційний пошук

Розглянувши лінійний, двійковий та пошук стрибками, ми завжди виходили з того, що розмір нашої колекції відомий. А що робити, якщо

масив настільки великий, що ми навіть не знаємо, де його кінець, або дані надходять у вигляді нескінченного потоку? Для таких випадків існує елегантний підхід — експоненційний пошук.

1. Теоретичні відомості

1.1. Ідея експоненційного пошуку: коли ми не знаємо, де кінець

Двійковий пошук є надзвичайно ефективним, але він має одну фундаментальну вимогу для своєї роботи: він повинен знати початкову та кінцеву межі діапазону пошуку. Що робити, якщо масив настільки великий, що його неможливо вмістити в пам'ять, або його розмір взагалі невідомий (наприклад, дані з нескінченного потоку)? У цьому випадку двійковий пошук застосувати неможливо, оскільки ми не можемо визначити середній елемент.

Експоненційний пошук (також відомий як пошук з подвоєнням або пошук Галопуючого) розв'язує саме цю проблему. Він поєднує швидкий пошук діапазону зі звичайним двійковим пошуком.

Аналогія: Уявіть, що ви загубили ключі на дуже довгій, прямій дорозі, яка йде за горизонт. Ви стоїте на її початку. Ви можете почати йти метр за метром (лінійний пошук), але це займе вічність. Значно розумніша стратегія — перевіряти відстань у кроках, що експоненційно зростають: спочатку ви дивитесь під ноги (позиція 1), потім на відстані 2 метри, потім 4 метри, 8, 16, і так далі, подвоюючи відстань на кожному кроці. Як тільки ви "перестрибнули" місце, де, ймовірно, лежать ключі (наприклад, на позначці 16 метрів ви нічого не бачите, але на 32 метрах ви впевнені, що зайшли занадто далеко), ви звузили область пошуку до керованого діапазону — від 16 до 32 метрів. І вже в цьому невеликому відрізку ви можете провести ретельний пошук.

Алгоритм працює у дві фази:

1. Крок 1: Пошук діапазону. Алгоритм знаходить невеликий діапазон, в якому гарантовано знаходиться шуканий елемент (якщо він взагалі існує). Це робиться шляхом перевірки елементів на індексах, що є степенями двійки: 1, 2, 4, 8, ..., 2^k. Цей процес триває, доки елемент $A[2^k]$ не виявиться більшим за шуканий ключ. У цей момент ми знаємо, що наш елемент, якщо він існує, знаходиться в діапазоні між попереднім та поточним стрибком, тобто між індексами $[2^{k-1}, 2^k]$.

2. Крок 2: Двійковий пошук. Після того, як ми визначили відносно невеликий діапазон, ми запускаємо на ньому звичайний двійковий пошук, який ми вже детально вивчили.

Цей двофазний підхід робить алгоритм ідеальним для роботи з необмеженими або дуже великими послідовностями.

1.2. Псевдокод та його аналіз

```
// Вхід: A - відсортований масив, key - значення, яке потрібно знайти
.
// Для цього псевдокоду ми припускаємо, що знаємо довжину 'n',
// але логіка легко адаптується до невідомої довжини.
ФУНКЦІЯ ExponentialSearch(Array A, key):
    n = довжина(A)

    // Базовий випадок: перевіряємо перший елемент окремо.
    ЯКЩО A[0] == key ТО
        ПОВЕРНУТИ 0
    КІНЕЦЬ ЯКЩО

    // Крок  1: Пошук діапазону.
    // Знаходимо верхню межу 'bound', подвоюючи її на кожному кроці.
    bound = 1
    ПОКИ bound < n І A[bound] <= key:
        bound = bound * 2
    КІНЕЦЬ ПОКИ

    // Крок  2: Двійковий пошук у знайденому діапазоні.
    // Нижня межа діапазону - це половина поточної 'bound'.
    // Верхня межа - або сама 'bound', або кінець масиву.
    left_bound = bound / 2
    right_bound = min(bound, n - 1)

    // Викликаємо вже знайомий нам двійковий пошук.
    ПОВЕРНУТИ BinarySearch_Iterative(A, key, left_bound, right_bound)
КІНЕЦЬ ФУНКЦІЇ

// Допоміжна функція двійкового пошуку (з попереднього розділу)
// яка тепер приймає межі для пошуку.
ФУНКЦІЯ BinarySearch_Iterative(Array A, key, left, right):
    ПОКИ left <= right:
        middle = left + (right - left) / 2
        ЯКЩО A[middle] == key ТО ПОВЕРНУТИ middle
        ЯКЩО A[middle] < key ТО left = middle + 1
        ІНАКШЕ right = middle - 1
```

```
    КІНЕЦЬ ПОКИ
    ПОВЕРНУТИ -1
КІНЕЦЬ ФУНКЦІЇ
```

1.3. Аналіз складності

Ефективність експоненційного пошуку визначається сумою вартостей двох його фаз. Припустимо, шуканий елемент знаходиться на позиції i.

Часова складність:

- Крок 1 (Пошук діапазону): Ми знаходимо таку межу **bound = 2^k**, що **2^(k-1) < i ≤ 2^k**. Щоб знайти це k, нам потрібно зробити k стрибків. З нерівності випливає, що k приблизно дорівнює $\log_2 i$. Отже, складність цієї фази — $O(\log i)$.
- Крок 2 (Двійковий пошук): Ми виконуємо двійковий пошук у діапазоні від 2^{k-1} до 2^k. Розмір цього діапазону становить $2^k - 2^{k-1} = 2^{k-1}$, що приблизно дорівнює $i/2$. Складність двійкового пошуку на масиві розміром i становить $O(\log i)$.
- Загальна складність: Сумарна складність є $O(\log i) + O(\log i) = O(\log i)$.

Важливо зазначити, що складність залежить від позиції елемента i, а не від загального розміру масиву n. Якщо елемент знаходиться на початку, пошук буде надзвичайно швидким. У найгіршому випадку, коли елемент знаходиться в кінці масиву (i ≈ n), складність стає $O(\log n)$, що дорівнює складності звичайного двійкового пошуку.

Просторова складність: Якщо для другої фази використовується ітеративна версія двійкового пошуку, алгоритм потребує лише декілька змінних, тому його просторова складність є константною: $O(1)$.

2. Область застосування

Експоненційний пошук є ідеальним вибором у специфічних умовах, де звичайний двійковий пошук не може бути застосований або є менш ефективним.

– Пошук у необмежених або “нескінченних” послідовностях: Це головна сфера застосування. Коли розмір даних невідомий, експоненційний пошук дозволяє спочатку ефективно знайти діапазон, а потім вже працювати в ньому.

– Дуже великі масиви: Якщо масив настільки великий, що навіть одна операція доступу до його середини є дорогою, а шуканий елемент, ймовірно, знаходиться на початку, експоненційний пошук може виграти у двійкового. Він знайде вузький діапазон `[0...bound]` і буде працювати лише в ньому, тоді як двійковий пошук одразу "стрибне" в далеку середину `n/2`.

3. Приклади застосування

– Пошук у лог-файлах: Уявіть відсортований за часом лог-файл, який постійно зростає. Щоб знайти запис за певною датою, можна використати експоненційний пошук, оскільки повний розмір файлу невідомий.

– Пошук першого входження: Знайти перше входження числа "1" у нескінченному відсортованому масиві, що складається з нулів, а потім одиниць (`{0, 0, ..., 0, 1, 1, ...}`).

– Системи з потоковою обробкою даних: Коли дані надходять потоком і не зберігаються повністю в пам'яті.

Переваги та недоліки

Переваги експоненційного пошуку полягають у його здатності працювати з даними невідомого розміру, що робить його унікальним інструментом. Його часова складність $O(\log i)$ є надзвичайно привабливою, особливо коли очікується, що шукані елементи знаходяться ближче до початку великої колекції. У таких сценаріях він може бути навіть швидшим за двійковий пошук.

Основні недоліки випливають з його природи. Як і двійковий пошук, він вимагає, щоб дані були відсортовані. Для звичайних масивів відомого розміру, де шуканий елемент може бути де завгодно, експоненційний пошук не пропонує значних переваг над двійковим і може бути трохи повільнішим через двофазну структуру. Його реалізація є дещо складнішою, ніж простий двійковий пошук.

Контрольні запитання

1. Опишіть основну проблему, яку вирішує експоненційний пошук і яку не може вирішити двійковий пошук.
2. Назвіть та опишіть дві основні фази роботи експоненційного пошуку.
3. Поясніть, чому часова складність алгоритму залежить від позиції шуканого елемента i, а не від загального розміру масиву n.

4. У якому випадку експоненційний пошук може бути швидшим за двійковий на масиві відомого розміру?
5. Чому, незважаючи на складність $O(\log i)$, експоненційний пошук не використовується замість двійкового у всіх випадках?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму експоненційного пошуку.

2. Мета

Практичне засвоєння двофазної логіки експоненційного пошуку та його застосування для пошуку в масивах.

3. Завдання

Створити програму, яка реалізує функцію експоненційного пошуку. Програма має продемонструвати знаходження діапазону та подальший виклик двійкового пошуку.

4. Опис

Програма повинна містити реалізацію експоненційного пошуку, яка всередині себе викликає раніше реалізовану функцію двійкового пошуку. Це гарний приклад композиції алгоритмів.

5. Порядок виконання

1. Підготуйте інструменти: Візьміть вашу реалізацію ітеративного двійкового пошуку з попередньої лабораторної роботи.
2. Реалізуйте функцію `ExponentialSearch`:
    - Перевірте перший елемент масиву.
    - Реалізуйте цикл `while` для знаходження верхньої межі `bound` шляхом її подвоєння. Додайте виведення на екран, щоб бачити, як зростає `bound`.
    - Визначте нижню та верхню межі для двійкового пошуку.
    - Викличте вашу функцію двійкового пошуку для знайденого діапазону.
3. Створіть тестовий масив: В основній частині програми створіть великий відсортований масив (напр., 100 елементів).
4. Протестуйте: Перевірте роботу, шукаючи елемент на початку масиву (напр., індекс 5), в середині (індекс 50) та в кінці (індекс 95).

Спостерігайте за виведенням, щоб побачити, які діапазони знаходить перша Крок алгоритму.

6. Результат

Додаток, що коректно знаходить елементи у відсортованому масиві за допомогою експоненційного пошуку.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією експоненційного та двійкового пошуку.
2. Продемонструвати роботу додатку, показавши, як визначаються межі для різних шуканих елементів.
3. Бути готовим відповісти на питання:
    - Пояснити, як ваш код обробляє випадок, коли шуканий елемент знаходиться за межами масиву.
    - Чому ми викликаємо двійковий пошук, а не лінійний, у другій фазі?
    - Якою буде складність алгоритму, якби ми використовували лінійний пошук у другій фазі?

Тести

1. Експоненційний пошук є найбільш доцільним для:

a) Невпорядкованих масивів.

b) Масивів невідомого або нескінченного розміру.

c) Зв’язаних списків.

d) Хеш-таблиць.

2. Яка часова складність експоненційного пошуку в найгіршому випадку для масиву розміром N?

a) $O(N)$

b) $O(\sqrt{N})$

c) $O(\log N)$

d) $O(N \log N)$

3. Перша Крок експоненційного пошуку служить для:

a) Сортування масиву.

b) Знаходження середнього елемента.

c) Визначення діапазону, в якому може знаходитись елемент.

d) Перевірки кожного елемента.

4. Після того, як експоненційний пошук знаходить діапазон `[bound/2, bound]`, він викликає:

a) Лінійний пошук.

b) Пошук стрибками.

c) Інтерполяційний пошук.

d) Двійковий пошук.

5. Складність експоненційного пошуку $O(\log i)$ (де i — позиція елемента) означає, що:

a) Алгоритм завжди працює за логарифмічний час від розміру всього масиву.

b) Алгоритм швидший, якщо елемент знаходиться ближче до кінця масиву.

c) Алгоритм швидший, якщо елемент знаходиться ближче до початку масиву.

d) Складність не залежить від позиції елемента.

Відповіді: 1-b, 2-c, 3-c, 4-d, 5-c

Тести підвищеної складності

1. У якій із наступних ситуацій експоненційний пошук буде найбільш ефективним і, ймовірно, перевершить стандартний двійковий пошук?

a) У невеликому масиві (до 100 елементів), де накладні витрати на дві фази пошуку мінімальні.

b) У дуже великому відсортованому масиві (мільйони елементів), де шуканий елемент з високою ймовірністю знаходиться на початку.

c) У невпорядкованому масиві, оскільки перша Крок швидко знаходить діапазон.

d) У масиві, де шуканий елемент знаходиться рівно посередині.

2. Часова складність експоненційного пошуку оцінюється як $O(\log i)$, де `i` — це індекс шуканого елемента. Що це означає на практиці?

a) Швидкість пошуку залежить від загальної кількості елементів `n` в масиві.

b) Пошук найшвидший для елементів, що знаходяться в середині масиву.

c) Час пошуку практично не залежить від загального розміру масиву, а лише від того, наскільки далеко від початку знаходиться шуканий елемент.

d) Складність алгоритму є лінійною відносно позиції елемента.

3. Якщо перша Крок експоненційного пошуку зупинилася, коли індекс `bound = 128` став більшим за шуканий ключ, в якому діапазоні індексів буде виконуватися двійковий пошук у другій фазі?

a) `[0, 128]`

b) `[127, 256]`

c) `[64, 127]`

d) `[0, 64]`

4. Вам потрібно знайти перший елемент, більший за 1000, у відсортованому масиві невідомого (потенційно нескінченного) розміру. Яка стратегія буде найефективнішою?

a) Використовувати лінійний пошук, оскільки розмір невідомий.

b) Використовувати стандартний двійковий пошук, припустивши, що розмір масиву — максимальне значення `int`.

c) Використовувати експоненційні стрибки, щоб знайти перший елемент, більший за 1000, а потім двійковим пошуком у попередньому діапазоні знайти точну границю.

d) Використовувати пошук стрибками з кроком $\sqrt{1000}$.

5. У чому полягає ключова відмінність між експоненційним пошуком та пошуком стрибками ($O(\sqrt{n})$)?

a) Експоненційний пошук не вимагає, щоб масив був відсортованим.

b) Пошук стрибками використовує двійковий пошук у другій фазі, а експоненційний — лінійний.

c) Експоненційний пошук використовує кроки, що подвоюються, що робить його ефективним для необмежених масивів ($O(\log i)$), тоді як пошук стрибками використовує фіксований крок ($\sqrt{n}$) і вимагає знання розміру масиву.

d) Експоненційний пошук завжди повільніший за пошук стрибками.

Відповіді

1. b) Бінарний пошук завжди починає з середини масиву (`n/2`). Якщо `n` дуже велике, а `key` знаходиться на початку, двійковий пошук зробить багато "стрибків" у далеку частину масиву. Експоненційний пошук швидко знайде вузький діапазон біля початку і виконає двійковий пошук лише в ньому, що буде значно швидше.

2. c) Це ключова властивість алгоритму. На відміну від $O(\log n)$, де `n` — загальний розмір, складність $O(\log i)$ показує, що алгоритм ідеально підходить для ситуацій, коли елементи, що часто шукаються, знаходяться на початку великої колекції.

3. c) Перша Крок знаходить першу степінь двійки `bound = 2^k`, для якої `A[bound] > key`. Попереднім кроком був `bound/2 = 2^(k-1)`. Отже, шуканий елемент гарантовано знаходиться в діапазоні `[bound/2, bound-1]`. Для `bound=128`, попередній крок був на `64`, отже діапазон пошуку — `[64, 127]`.

4. c) Це пряме застосування алгоритму. Експоненційні стрибки (`1, 2, 4, ...`) дозволяють за логарифмічний час знайти верхню межу, де знаходиться шукана зміна значення. Потім двійковий пошук точно локалізує перший такий елемент. Стандартний двійковий пошук неможливий без знання кінця масиву.

5. c) Це фундаментальна відмінність у стратегії. Експоненційне зростання кроку дозволяє експоненційному пошуку ефективно працювати з невідомими розмірами і дає логарифмічну складність відносно позиції елемента. Пошук стрибками прив'язаний до загального розміру `n` і має гіршу складність.

Завдання для самостійної роботи

1. Рекурсивна реалізація. Напишіть повністю рекурсивну версію експоненційного пошуку (де і перша, і друга Крок реалізовані рекурсивно).
2. Обробка відсутності елемента. Проаналізуйте та детально опишіть, як псевдокод обробляє випадок, коли шуканий елемент `key` більший за всі елементи в масиві.
3. Зміна основи. Модифікуйте алгоритм, щоб він використовував стрибки зі зростанням по основі 3 (1, 3, 9, 27, ...) замість 2. Як це теоретично вплине на асимптотичну складність?

4. Порівняння кількості операцій. Візьміть великий масив (N=1024). Порівняйте кількість порівнянь, яку зроблять двійковий та експоненційний пошук для знаходження елемента на позиції 10 та на позиції 1000.
5. Перше входження. Напишіть версію експоненційного пошуку, яка гарантовано знаходить індекс першого входження елемента в масив, де можуть бути дублікати.
6. Пошук у потоці. Опишіть, як би ви реалізували експоненційний пошук, якби дані надходили з “потоку”, де ви не можете дізнатися довжину і можете лише послідовно зчитувати елементи.
7. Недоліки для зв’язаних списків. Поясніть, чому експоненційний пошук (як і двійковий) є неефективним для застосування до зв’язаних списків, навіть якщо вони відсортовані.
8. Нижня межа. Для відсортованого масиву, що починається з великої кількості від’ємних чисел, а потім містить додатні, розробіть алгоритм пошуку першого невід’ємного елемента на основі ідеї експоненційного пошуку.
9. Есе-порівняння. Напишіть есе, що порівнює пошук стрибками ($O(\sqrt{n})$) та експоненційний пошук ($O(\log n)$). Чому, незважаючи на схожість ідеї “стрибків”, їхня складність кардинально відрізняється?
10. Гібридний підхід. Запропонуйте та опишіть гібридний алгоритм, який для масивів до певного розміру `MAX_SIZE` використовує двійковий пошук, а для більших — експоненційний. Обґрунтуйте, чи має такий підхід сенс.

Гібридні алгоритми пошуку

Інтерполяційно-двійковий пошук

1. Теоретичні відомості

1.1. Ідея гібридного підходу: страхування від найгіршого випадку

У попередніх розділах ми дослідили два потужні алгоритми пошуку у відсортованих масивах:

– Двійковий пошук: Надійний та стабільний “робочий кінь”. Він завжди гарантує знаходження елемента за час $O(\log n)$, незалежно від характеру даних, “сліпо” перевіряючи середину діапазону.

– Інтерполяційний пошук: Інтелектуальний “спеціаліст”. Для рівномірно розподілених даних він демонструє феноменальну середню швидкість $O(\log(\log n))$. Однак, для “поганих”, нерівномірних даних, його продуктивність катастрофічно падає до $O(n)$.

Це створює для інженера дилему: ризикнути заради потенційно вищої швидкості інтерполяційного пошуку чи обрати гарантовану стабільність двійкового? Гібридний підхід пропонує не обирати, а поєднати.

Основна ідея: Почати з найшвидшого методу (інтерполяційного), але мати “план Б”. Ми будемо використовувати інтерполяційний пошук, але водночас уважно стежити за його продуктивністю. Якщо ми помічаємо, що він починає працювати погано (тобто, дані, скоріше за все, нерівномірні), ми негайно перемикаємось на надійний двійковий пошук, щоб завершити роботу.

Це як мати гоночний болід та надійний позашляховик. Ви починаєте гонку на боліді, але якщо бачите, що попереду починається бездоріжжя, ви пересідаєте на позашляховик, щоб гарантовано дістатися до фінішу.

1.2. Критерій перемикання: коли відмовлятись від інтерполяції

Це ключове питання гібридного алгоритму: як саме зрозуміти, що інтерполяція “не справляється”? Існує декілька евристик, що дозволяють це визначити.

1. Вихід за межі діапазону. Формула інтерполяції

   ```
   pos = left + ((key - A[left]) * (right - left)) / (A[right] - A[left])
   ```

   дуже чутлива до розподілу. Якщо дані нерівномірні, обчислений індекс `pos` може легко вийти за межі поточного діапазону пошуку `[left, right]`. Це найнадійніший сигнал про те, що дані “погані” і потрібно перемикатися.

2. Занадто повільна збіжність. Двійковий пошук гарантовано скорочує діапазон пошуку приблизно вдвічі на кожному кроці. “Здоровий” інтерполяційний пошук має робити це значно швидше. Ми можемо встановити правило: якщо після одного або двох кроків інтерполяції розмір діапазону (`right - left`) скоротився менше, ніж удвічі, це привід для тривоги і перемикання на двійковий пошук.

Зазвичай використовується перший, простіший критерій, оскільки він надійно виявляє проблемні дані.

1.3. Псевдокод гібридного алгоритму

```
// Вхід: A - відсортований масив, key - значення, яке потрібно знайти
.
// Вихід: індекс знайденого елемента, або -1, якщо не знайдено.
ФУНКЦІЯ HybridInterpolationBinarySearch(Array A, key):
    left = 0
    right = довжина(A) - 1

    ПОКИ left <= right І key >= A[left] І key <= A[right]:
        // Крок 1: Обчислюємо ймовірну позицію за формулою інтерполяц
ії
        // (Перевірка A[right] != A[left] потрібна, щоб уникнути діле
ння на нуль)
        ЯКЩО A[right] == A[left] ТО
            ЯКЩО A[left] == key ТО ПОВЕРНУТИ left ІНАКШЕ ПОВЕРНУТИ -1
КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ЯКЩО

        pos = left + ціла_частина(((key - A[left]) * (right - left))
/ (A[right] - A[left]))

        // Крок 2: Критерій перемикання.
        // Якщо позиція вийшла за межі, дані, ймовірно, нерівномірні.
        ЯКЩО pos < left АБО pos > right ТО
            // Перемикаємось на надійний двійковий пошук для поточног
о діапазону
            ПОВЕРНУТИ BinarySearch_Iterative(A, key, left, right)
        КІНЕЦЬ ЯКЩО

        // Крок 3: Стандартна логіка, як в інтерполяційному пошуку
        ЯКЩО A[pos] == key ТО
            ПОВЕРНУТИ pos
        КІНЕЦЬ ЯКЩО

        ЯКЩО A[pos] < key ТО
            left = pos + 1
        ІНАКШЕ
            right = pos - 1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ

    ПОВЕРНУТИ -1
КІНЕЦЬ ФУНКЦІЇ
```

```
// Допоміжна функція двійкового пошуку, яка викликається у випадку "а
варії"
ФУНКЦІЯ BinarySearch_Iterative(Array A, key, left, right):
    ПОКИ left <= right:
        middle = left + (right - left) / 2
        ЯКЩО A[middle] == key ТО ПОВЕРНУТИ middle
        ЯКЩО A[middle] < key ТО left = middle + 1
        ІНАКШЕ right = middle - 1
    КІНЕЦЬ ПОКИ
    ПОВЕРНУТИ -1
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Аналіз складності

Гібридний підхід дозволяє нам отримати найкраще з обох світів.

- Часова складність (середній випадок): Якщо дані розподілені рівномірно, критерій перемикання ніколи не спрацює, і алгоритм буде поводитися як чистий інтерполяційний пошук. Складність: $O(\log(\log n))$.
- Часова складність (найгірший випадок): Якщо дані розподілені нерівномірно, алгоритм після кількох перших кроків виявить це і перемкнеться на двійковий пошук. Подальша робота буде домінуватися саме двійковим пошуком. Таким чином, ми замінили катастрофічний найгірший випадок $O(n)$ на дуже ефективний. Складність: $O(\log n)$.
- Просторова складність: Алгоритм є ітеративним і використовує константну кількість додаткової пам'яті. Просторова складність: $O(1)$.

2. Область застосування

Гібридний інтерполяційно-двійковий пошук — це саме та реалізація, яка робить інтерполяційний пошук життєздатним на практиці. Він використовується в ситуаціях, коли:

- Очікується, що дані будуть рівномірно розподілені, але потрібна гарантія стабільної продуктивності для будь-яких даних.
- Розробляються високопродуктивні бібліотечні функції, які мають працювати надійно з будь-якими вхідними даними.
- Пошук виконується над дуже великими масивами, де виграш від $O(\log(\log n))$ може бути суттєвим, але ризик $O(n)$ є неприпустимим.

3. Приклади застосування

– Оптимізовані бібліотеки: Коли ви бачите в документації до стандартної бібліотеки згадку про “інтерполяційний пошук”, найчастіше мається на увазі саме така робастна гібридна реалізація.
– Системи баз даних: При пошуку по індексу, де значення ключів очікуються бути рівномірно розподіленими (наприклад, ID, що генеруються послідовно), але можливі аномалії.

Переваги та недоліки

Основна перевага гібридного алгоритму полягає в його адаптивності та надійності. Він поєднує видатну швидкість інтерполяційного пошуку для ідеальних даних з “залізною” гарантією продуктивності двійкового пошуку для найгірших випадків. Це дозволяє використовувати його в реальних системах без страху отримати непередбачуване падіння продуктивності.

До недоліків можна віднести дещо більшу складність реалізації порівняно з кожним з його компонентів окремо. На кожній ітерації виконується додаткова перевірка (критерій перемикання), що створює невеликі накладні витрати. Для малих масивів або у випадках, де дані гарантовано рівномірні, чистий інтерполяційний пошук може бути трохи швидшим.

Контрольні запитання

1. У чому полягає головна мета створення гібридного інтерполяційно-двійкового пошуку?
2. Опишіть один із можливих критеріїв, за яким гібридний алгоритм приймає рішення перемкнутися з інтерполяційного на двійковий пошук.
3. Порівняйте часову складність у найгіршому випадку для чистого інтерполяційного пошуку та для гібридного.
4. Чи має сенс створювати гібридний лінійно-двійковий пошук? Чому?
5. Як зміниться поведінка гібридного алгоритму, якщо порогове значення для перемикання зробити занадто “чутливим” (тобто, перемикатися при найменшій нерівномірності)?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація гібридного інтерполяційно-двійкового пошуку.

2. Мета

Практичне засвоєння техніки побудови гібридних алгоритмів. Реалізація критерію перемикання та аналіз поведінки алгоритму на різних типах даних.

3. Завдання

Створити програму, яка реалізує функцію гібридного пошуку. Провести серію експериментів, щоб продемонструвати, як алгоритм поводиться на рівномірно та нерівномірно розподілених даних.

4. Опис

Програма повинна містити реалізацію гібридного алгоритму, яка всередині себе може викликати реалізацію двійкового пошуку.

5. Порядок виконання

1. Підготуйте інструменти: Візьміть вашу реалізацію ітеративного двійкового пошуку з попередніх лабораторних робіт.
2. Реалізуйте гібридну функцію: Напишіть функцію `HybridSearch`, керуючись наведеним псевдокодом.
3. Реалізуйте критерій перемикання: Всередині циклу `while` додайте умову `if`, яка перевіряє, чи виходить обчислена позиція `pos` за межі `[left, right]`.
4. Створіть генератори даних:
    - Напишіть функцію, що генерує рівномірно розподілений відсортований масив.
    - Напишіть функцію, що генерує нерівномірно розподілений відсортований масив (напр., експоненційний).
5. Проведіть експеримент:
    - Запустіть гібридний пошук на рівномірно розподіленому масиві. Додайте виведення на екран, щоб переконатись, що перемикання на двійковий пошук не відбувається.
    - Запустіть гібридний пошук на нерівномірно розподіленому масиві. Прослідкуйте, як на певному кроці спрацьовує критерій і відбувається перемикання.

6. Результат

Додаток, що реалізує гібридний пошук і наочно демонструє його адаптивну поведінку на різних наборах даних.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією гібридного та двійкового пошуку.
2. Продемонструвати роботу додатку, показавши, як і коли відбувається перемикання на двійковий пошук на нерівномірних даних.
3. Бути готовим відповісти на питання:
    - Пояснити логіку вашого критерію перемикання.
    - Яка фінальна складність вашого алгоритму в найгіршому випадку і чому?
    - Чи можна було б перемикатися не на двійковий, а на пошук стрибками? Чи мало б це сенс?

Тести

1. Яку проблему чистого інтерполяційного пошуку вирішує гібридний підхід?

a) Неможливість роботи з від'ємними числами.

b) Високу просторову складність $O(n)$.

c) Деградацію часової складності до $O(n)$ на нерівномірних даних.

d) Повільну роботу на малих масивах.

2. Яка часова складність гібридного інтерполяційно-двійкового пошуку в найгіршому випадку?

a) $O(n)$

b) $O(\sqrt{n})$

c) $O(\log n)$

d) $O(\log(\log n))$

3. Критерій перемикання на двійковий пошук в гібридному алгоритмі зазвичай спрацьовує, коли:

a) Масив не відсортований.

b) Шуканий елемент не знайдено.

c) Розмір масиву стає меншим за 10.

d) Обчислена за формулою інтерполяції позиція виходить за межі поточного діапазону пошуку.

4. Для великого, ідеально рівномірно розподіленого масиву, гібридний алгоритм буде поводитись як:

a) Чистий двійковий пошук.

b) Чистий інтерполяційний пошук.

c) Лінійний пошук.

d) Пошук стрибками.

5. Головна перевага гібридного підходу полягає у:

a) Простоті реалізації.

b) Найкращій асимптотичній складності серед усіх алгоритмів пошуку.

c) Поєднанні швидкості для середнього випадку та надійності для найгіршого випадку.

d) Мінімальному використанні пам’яті.

Відповіді: 1-c, 2-c, 3-d, 4-b, 5-c

Тести підвищеної складності

1. У гібридному інтерполяційно-двійковому пошуку, чому обчислення індексу `pos`, що виходить за межі поточного діапазону `[left, right]`, є надійним сигналом для переходу до двійкового пошуку?

a) Це означає, що шуканий елемент гарантовано відсутній у масиві.

b) Це вказує на те, що масив не відсортований, і подальший пошук неможливий.

c) Це є ознакою потенційного цілочисельного переповнення у формулі.

d) Це свідчить про те, що припущення про лінійний розподіл даних сильно порушено, і формула інтерполяції дає некоректний результат.

2. Для якого з наступних наборів даних гібридний пошук, найімовірніше, покаже майже таку ж кількість порівнянь, як і чистий двійковий пошук?

a) `{10, 20, 30, 40, ..., 10000}` (рівномірний розподіл).

b) `{1, 2, 4, 8, 16, 32, ..., 2^N}` (експоненційний розподіл).

c) Дуже малий масив (N < 16).

d) Масив, де всі елементи однакові.

3. Коли гібридний алгоритм приймає рішення перемкнутися на двійковий пошук, які параметри він має передати у функцію двійкового пошуку для коректного продовження роботи?

a) Тільки оригінальний масив та ключ, розпочавши новий пошук з нуля.

b) Лише підмасив від `left` до `right`, створивши його копію.

c) Оригінальний масив, ключ, а також поточні значення меж `left` та `right` з фази інтерполяції.

d) Тільки останню обчислену позицію `pos` та ключ.

4. Ідею гібридного підходу (швидкий, але нестабільний метод + надійний "запасний" варіант) можна застосувати і до сортування. Яка з наступних пар є найкращим прикладом такої гібридизації?

a) Сортування злиттям + Сортування бульбашкою.

b) Швидке сортування (QuickSort) + Пірамідальне сортування (HeapSort).

c) Сортування включенням + Сортування вибором.

d) Сортування за розрядами (Radix Sort) + Сортування підрахунком (Counting Sort).

5. Ви розробляєте прошивку для мікроконтролера з дуже обмеженою пам'яттю програм (Flash). Вам потрібна функція пошуку для відсортованого масиву, дані в якому майже завжди розподілені рівномірно. Який алгоритм буде найкращим вибором з точки зору балансу швидкості та розміру скомпільованого коду?

a) Чистий інтерполяційний пошук, оскільки він має найкращу середню складність $O(\log(\log n))$.

b) Гібридний пошук, оскільки він найшвидший та найнадійніший.

c) Лінійний пошук, оскільки він має найменший розмір коду.

d) Чистий двійковий пошук, оскільки він пропонує відмінну швидкість $O(\log n)$ при значно простішій та компактнішій реалізації, ніж гібридний пошук.

Відповіді

1. d) Формула інтерполяції працює як лінійне наближення. Якщо дані нелінійні, "екстраполяція" може легко вивести індекс за межі відомого діапазону. Це є чітким індикатором того, що основне припущення алгоритму не виконується.

2. b) На експоненційно розподілених даних інтерполяційний пошук показує найгіршу продуктивність. Гібридний алгоритм майже одразу це виявить (на 1-2 ітерації) і перемкнеться на двійковий пошук, який і виконає переважну більшість роботи. Тому їхня продуктивність буде майже ідентичною.

3. c) Двійковий пошук має продовжити роботу з того місця, де зупинився інтерполяційний. Передавши поточні межі `left` та `right`, ми гарантуємо, що пошук продовжиться у вже звуженому діапазоні, не втрачаючи прогресу.

4. b) Це принцип, що лежить в основі IntroSort, який є стандартним алгоритмом сортування в багатьох бібліотеках. Він використовує швидкий QuickSort, але якщо глибина рекурсії стає занадто великою (ознака найгіршого випадку $O(n^2)$), він перемикається на гарантовано стабільний HeapSort ($O(n\log n)$).

5. d) У системах з жорсткими обмеженнями на розмір коду, складність реалізації має велике значення. Гібридний алгоритм вимагає коду для обох методів та логіки перемикання. Двійковий пошук пропонує чудову та гарантовану продуктивність при набагато меншому та простішому коді, що є кращим компромісом у даному сценарії.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте гібридний алгоритм згідно з методичними вказівками.
2. Альтернативний критерій. Модифікуйте ваш гібридний алгоритм, додавши інший критерій перемикання: якщо за два кроки інтерполяції діапазон пошуку скоротився менше, ніж на 25%, перемикатися на двійковий пошук.
3. Емпіричний аналіз. Проведіть експеримент, щоб порівняти кількість порівнянь для чистого двійкового, чистого інтерполяційного та гібридного пошуку на трьох наборах даних: рівномірному, експоненційному та “скупченому” (напр., `{1, 2, 3, ..., 100, 1000, 10000}`).
4. Гібрид з пошуком стрибками. Поміркуйте, чи має сенс створювати гібридний пошук, що перемикається з інтерполяційного на пошук стрибками. Обґрунтуйте свою відповідь.
5. Гібридний пошук для `find first`. Адаптуйте гібридний алгоритм для задачі пошуку першого входження елемента в масив з дублікатами.

6. Аналіз порогу. Як вибір порогового значення в лабораторній роботі впливає на продуктивність гібридного алгоритму? Проведіть експерименти з різними значеннями.
7. Пошук кореня функції. Дослідіть, як ідея гібридизації застосовується в чисельних методах, наприклад, поєднання методу хорд (аналог інтерполяції) та методу бісекції (аналог двійкового пошуку) для знаходження кореня функції.
8. Адаптивний вибір. Розробіть алгоритм, який на початку аналізує невелику вибірку з масиву, щоб визначити, чи є дані рівномірними, і на основі цього одразу обирає або двійковий, або інтерполяційний пошук.
9. Есе-порівняння. Напишіть есе, що порівнює філософію “завжди надійного” двійкового пошуку та “адаптивного, але складнішого” гібридного підходу з точки зору інженерії ПЗ.
10. Зворотна задача. Створіть генератор масивів, який цілеспрямовано “ламає” інтерполяційний пошук, змушуючи його деградувати до $O(n)$.
11. Тернарний пошук. Чи можна створити гібридний інтерполяційно-тернарний пошук? Для яких задач це могло б мати сенс?
12. Робота з рядками. Подумайте, як можна було б адаптувати гібридний пошук для роботи з відсортованим масивом довгих рядків.
13. Аналіз реальних бібліотек. Дослідіть документацію стандартної бібліотеки вашої улюбленої мови програмування. Чи використовується в ній інтерполяційний або гібридний пошук для якихось функцій?
14. Просторова складність. Детально проаналізуйте просторову складність рекурсивної версії гібридного пошуку.
15. Есе-роздум. Напишіть есе на тему: “Гібридні алгоритми як приклад інженерного компромісу між теоретичною оптимальністю та практичною надійністю”.

Гібридний двійково-лінійний пошук

1. Теоретичні відомості

1.1. Ідея гібридного підходу: коли складність — це зайві витрати

Ми встановили, що двійковий пошук з його складністю $O(\log n)$ є беззаперечним чемпіоном з ефективності для пошуку у великих відсортованих масивах. Він елегантний, швидкий і є втіленням парадигми “Розділяй і володарюй”. Але чи завжди він є найкращим вибором на практиці, особливо коли діапазон пошуку стає дуже малим?

Аналогія: Уявіть, що ви шукаєте потрібний документ у величезному архіві з мільйоном папок. Використовувати складну систему каталогів (двійковий пошук) — це найшвидший спосіб знайти потрібний стелаж. Але коли ви звузили пошук до однієї полиці, на якій лежить всього 10 папок, чи будете ви продовжувати ділити цю маленьку стопку навпіл? Скоріше за все, ви просто швидко переглянете їх усі по черзі (лінійний пошук).

Ця інтуїція лежить в основі гібридного двійково-лінійного пошуку. Ідея полягає в тому, що кожен алгоритм має не тільки асимптотичну складність, але й накладні витрати. Для двійкового пошуку це:

- Обчислення середнього індексу на кожній ітерації.
- Підтримка меж `left` та `right`.
- Витрати на рекурсивний виклик (якщо реалізація рекурсивна).

Для дуже малих масивів (наприклад, 10-20 елементів) ці накладні витрати можуть переважити вигоду від логарифмічного скорочення пошуку. Простий лінійний перебір на такому короткому відрізку може виявитись швидшим на практиці, оскільки його логіка є гранично простою для процесора.

1.2. Визначення порогу перемикання

Гібридний алгоритм поєднує сильні сторони обох методів:

3. Він використовує швидкий двійковий пошук для значного скорочення великих діапазонів.
4. Коли діапазон пошуку стає достатньо малим, він перемикається на простий та ефективний для малих даних лінійний пошук.

Ключовим елементом цього підходу є порогове значення (THRESHOLD). Це константа, що визначає розмір масиву, нижче якого ми відмовляємось від двійкового пошуку на користь лінійного.

Як обрати порогове значення? Важливо розуміти, що не існує єдиного “магічного” числа для **THRESHOLD**. Його оптимальне значення є емпіричним і залежить від:

- Мови програмування та її компілятора/інтерпретатора.
- Архітектури конкретного процесора.
- Вартості операцій порівняння для конкретного типу даних.

Зазвичай це значення лежить в діапазоні від 8 до 64 і визначається шляхом практичних тестів для конкретної системи.

1.3. Псевдокод гібридного алгоритму

```
// Константа, що визначає поріг перемикання
КОНСТАНТА THRESHOLD = 16

// Вхід: A - відсортований масив, key - ключ для пошуку.
// Вихід: індекс знайденого елемента, або -1.
ФУНКЦІЯ HybridBinaryLinearSearch(Array A, key):
    left = 0
    right = довжина(A) - 1

    // Крок  1: Двійковий пошук, поки діапазон великий.
    ПОКИ (right - left + 1) > THRESHOLD:
        middle = left + (right - left) / 2
        ЯКЩО A[middle] < key ТО
            left = middle + 1
        ІНАКШЕ
            right = middle
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ

    // Крок  2: Лінійний пошук на малому залишковому діапазоні [left,
right].
    ДЛЯ i ВІД left ДО right:
        ЯКЩО A[i] == key ТО
            ПОВЕРНУТИ i
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    ПОВЕРНУТИ -1
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Аналіз складності

– Часова складність: Це дуже важливий момент. Гібридизація з лінійним пошуком є мікрооптимізацією, яка не змінює асимптотичну складність.

1. Основна частина роботи виконується у фазі двійкового пошуку, яка скорочує діапазон з n до константного розміру `THRESHOLD`. Ця частина має складність $O(\log n)$.
2. Друга Крок — лінійний пошук — виконується на масиві, розмір якого не перевищує `THRESHOLD`. Оскільки `THRESHOLD` є константою, складність цієї фази — $O(\text{THRESHOLD}) = O(1)$.
3. Загальна складність є сумою цих двох частин: $O(\log n) + O(1) = O(\log n)$. Таким чином, асимптотично алгоритм залишається логарифмічним.

– Просторова складність: Алгоритм є ітеративним і використовує константну кількість додаткових змінних. Просторова складність: $O(1)$.

2. Область застосування

Цей гібридний підхід є класичним прикладом практичної оптимізації, яка застосовується не стільки в прикладному коді, скільки в реалізаціях низькорівневих, високопродуктивних бібліотек.

– Стандартні бібліотеки мов програмування: Реалізації функцій пошуку (`std::lower_bound` в C++) або сортування (де на малих підмасивах QuickSort може перемикатися на Insertion Sort) часто використовують подібні гібридні техніки.
– Критичні до продуктивності системи: У системах реального часу, ігрових рушіях, наукових обчисленнях, де кожна мікросекунда має значення.

3. Приклади застосування

– Реалізація `TimSort`: Стандартний алгоритм сортування в Python та Java є гібридним. Він використовує сортування вставками (яке дуже ефективне на малих та частково впорядкованих масивах) для невеликих блоків даних.
– Бібліотеки для наукових обчислень: При роботі з великими векторами та матрицями багато функцій оптимізовані з урахуванням порогових значень, нижче яких використовуються простіші алгоритми.

Переваги та недоліки

Переваги гібридного двійково-лінійного пошуку полягають у його практичній продуктивності. Завдяки усуненню накладних витрат двійкового пошуку на малих діапазонах, він може працювати на декілька відсотків швидше за чисту версію. Це приклад того, як теоретичний аналіз доповнюється емпіричним для досягнення найкращого реального результату.

Недоліки також очевидні. По-перше, реалізація стає дещо складнішою. По-друге, виграш у продуктивності є невеликим і помітний лише при дуже великій кількості операцій пошуку. По-третє, оптимальне значення `THRESHOLD` є залежним від платформи, що вимагає додаткових тестів та налаштувань для кожної системи, на якій буде виконуватись код.

Контрольні запитання

1. У чому полягає основна ідея гібридного двійково-лінійного пошуку?
2. Що таке “накладні витрати” алгоритму і як вони впливають на вибір між двійковим та лінійним пошуком для малих масивів?
3. Поясніть, чому гібридизація з лінійним пошуком не змінює загальну асимптотичну складність $O(\log n)$.
4. Від яких факторів залежить вибір оптимального порогового значення (`THRESHOLD`)?
5. Чи має сенс створювати гібридний пошук стрибками-лінійний пошук? Обґрунтуйте.

Методичні вказівки до лабораторної роботи

1. Назва

Емпіричне визначення оптимального порогу перемикання для гібридного двійково-лінійного пошуку.

2. Мета

Практично дослідити вплив накладних витрат на продуктивність алгоритмів. Навчитись проводити бенчмаркінг для визначення оптимальних параметрів гібридного алгоритму.

3. Завдання

Створити програму, яка реалізує гібридний двійково-лінійний пошук. Провести серію експериментів з різними значеннями `THRESHOLD`, щоб знайти оптимальне для вашої системи.

4. Опис

Програма має містити реалізацію гібридного алгоритму. Основна частина буде запускати пошук велику кількість разів для фіксованого масиву, але з різними значеннями порогу перемикання, і вимірювати середній час виконання.

5. Порядок виконання

1. Реалізуйте алгоритми: Напишіть функцію лінійного пошуку та функцію гібридного пошуку, яка приймає `THRESHOLD` як параметр.
2. Створіть тестовий набір: Згенеруйте великий відсортований масив (напр., 1,000,000 елементів).
3. Напишіть функцію для тестування: Ця функція має приймати значення `THRESHOLD`, виконувати пошук 100,000 разів (для випадкових ключів) і повертати загальний час виконання.
4. Проведіть експеримент: Запустіть вашу тестову функцію в циклі, змінюючи `THRESHOLD` від, наприклад, 2 до 100.
5. Знайдіть оптимум: Виведіть результати у вигляді “Поріг: X, Час: Y”. Знайдіть значення порогу, при якому час виконання був мінімальним.

6. Результат

Програма, що дозволяє емпірично визначити оптимальний поріг перемикання, та отримане значення цього порогу для вашої системи.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми.
2. Результати вимірювань та знайдене оптимальне значення `THRESHOLD`.
3. Бути готовим відповісти на питання:
    - Чому графік залежності часу від порогу, ймовірно, матиме U-подібну форму?
    - Як змінилися б результати, якби ви використовували іншу мову програмування або інший компілятор?
    - Поясніть логіку вашого гібридного алгоритму.

Тести

1. Гібридний двійково-лінійний пошук перемикається на лінійний, коли:

a) Елемент не знайдено.

b) Масив не відсортований.

c) Діапазон пошуку стає достатньо малим.

d) Значення шуканого ключа дуже велике.

2. Яка асимптотична часова складність гібридного двійково-лінійного пошуку?

a) $O(n)$

b) $O(\sqrt{n})$

c) $O(\log n)$

d) $O(1)$

3. Оптимальне значення порогу перемикання (`THRESHOLD`) зазвичай:

a) Завжди дорівнює 10.

b) Обчислюється за формулою $\sqrt{n}$.

c) Визначається експериментально (емпірично).

d) Дорівнює $\log_2 n$.

4. Основна перевага гібридного двійково-лінійного пошуку полягає у:

a) Зменшенні асимптотичної складності.

b) Можливості працювати на невпорядкованих даних.

c) Зменшенні накладних витрат на малих діапазонах пошуку.

d) Зменшенні вимог до пам’яті.

5. Для якого завдання такий гібридний підхід є найбільш виправданим?

a) Для одноразового пошуку в малому масиві.

b) Для реалізації високопродуктивної бібліотечної функції, що буде викликатись мільйони разів.

c) Для пошуку у зв’язаному списку.

d) Для написання максимально простого і зрозумілого коду.

Відповіді 1-c, 2-c, 3-c, 4-c, 5-b

Тести підвищеної складності

1. Розглянемо **`IntroSort`** — гібридний алгоритм сортування. Він використовує Швидке сортування, але перемикається на Пірамідальне, коли глибина рекурсії стає завеликою, і на Сортування

включенням для малих розділів. Який недолік якого алгоритму він намагається "застрахувати", перемикаючись на Пірамідальне сортування?

a) Високі вимоги до пам'яті у Швидкого сортування.

b) Нестабільність Пірамідального сортування.

c) Найгірший випадок Швидкого сортування зі складністю $O(n^2)$.

d) Повільну роботу Сортування включенням на великих масивах.

2. Ви реалізували гібридний пошук і емпірично визначили, що для вашої системи оптимальний `THRESHOLD` = 32. Що станеться, якщо ви встановите `THRESHOLD` = 1000?

a) Алгоритм стане значно швидшим, бо лінійний пошук буде використовуватись частіше.

b) Алгоритм стане повільнішим, оскільки для великих діапазонів (напр., від 33 до 1000) він буде використовувати менш ефективний лінійний пошук замість двійкового.

c) Нічого не зміниться, оскільки поріг є лише теоретичним поняттям.

d) Алгоритм перестане працювати для масивів розміром менше 1000.

3. Чому гібридний підхід "інтерполяційний → двійковий" є концептуально більш значущим, ніж "двійковий → лінійний"?

a) Тому що він простіший в реалізації.

b) Тому що він змінює асимптотичну складність у найгіршому випадку (з $O(n)$ до $O(\log n)$), тоді як другий є лише мікрооптимізацією.

c) Тому що він не вимагає сортування даних.

d) Тому що він використовує менше пам'яті.

4. Ви пишете функцію пошуку для 64-бітного цілочисельного масиву. Який фактор, окрім логіки алгоритму, може вплинути на вибір оптимального `THRESHOLD` порівняно з 8-бітним масивом символів?

a) Наявність від'ємних чисел.

b) Вартість операції порівняння: порівняння 64-бітних чисел може бути дещо дорожчим, що може трохи збільшити оптимальний поріг.

c) Загальний розмір масиву в байтах.

d) Жоден фактор не вплине, поріг залежить лише від кількості елементів.

5. Яке твердження найбільш точно описує філософію гібридних алгоритмів?

a) Завжди обирати алгоритм з найкращою асимптотичною складністю.

b) Поєднувати різні алгоритми для досягнення найкращого балансу між продуктивністю в середньому, продуктивністю в найгіршому випадку та накладними витратами.

c) Писати максимально простий код, ігноруючи продуктивність.

d) Використовувати рекурсію замість ітерації для покращення читабельності.

Відповіді на тести підвищеної складності

1. c) **`IntroSort`** використовує **`HeapSort`** як "страховку". Якщо **`QuickSort`** починає "вироджуватись" і наближатись до своєї найгіршої квадратичної складності, IntroSort перемикається на **`HeapSort`**, який гарантує складність $O(nlogn)$ за будь-яких умов.

2. b) Ідея гібриду — використовувати лінійний пошук лише для дуже малихдіапазонів. Завищений поріг змушує алгоритм застосовувати повільний лінійний пошук там, де значно ефективнішим був би двійковий, що погіршує загальну продуктивність.

3. b) Гібрид "інтерполяційний → двійковий" — це фундаментальне покращення, яке рятує від катастрофічного найгіршого випадку $O(n)$. Гібрид "двійковий → лінійний" не змінює загальну складність $O(\log n)$, а лише трохи покращує реальну швидкість за рахунок усунення накладних витрат на малих даних.

4. b) Оптимальний поріг — це баланс між накладними витратами алгоритму та вартістю порівняння. Якщо порівняння є "дорожчим" (як для 64-бітних чисел), вигода від зменшення їх кількості (що робить двійковий пошук) залишається суттєвою для більших діапазонів. Це дозволяє "терпіти" накладні витрати двійкового пошуку довше, тому поріг переходу на лінійний пошук буде вищим.

5. b) Це квінтесенція інженерного підходу. Гібридні алгоритми не шукають один "ідеальний" метод, а створюють адаптивну систему, яка використовує сильні сторони різних алгоритмів (швидкість у середньому випадку, надійність у найгіршому, ефективність на малих даних) для досягнення найкращого практичного результату.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте гібридний двійково-лінійний пошук згідно з методичними вказівками.
2. Емпіричний аналіз. Проведіть експеримент для визначення оптимального `THRESHOLD` на вашому комп'ютері та для вашої мови програмування.
3. Гібридний пошук стрибками. Розробіть гібридний алгоритм "пошук стрибками → лінійний пошук". Чи має він сенс? (Підказка: сам пошук стрибками вже є таким гібридом).
4. Аналіз коду бібліотек. Дослідіть вихідний код реалізації функції сортування у стандартній бібліотеці вашої мови (напр., `std::sort` в C++). Знайдіть, чи використовує вона гібридний підхід.
5. Порівняння реалізацій. Порівняйте продуктивність вашої гібридної функції з вбудованою функцією пошуку (якщо така є).
6. Гібридний QuickSort. Реалізуйте Швидке сортування, яке перемикається на Сортування включенням, коли розмір підмасиву стає меншим за певний поріг.
7. Аналіз впливу типу даних. Проведіть експеримент: чи зміниться оптимальний `THRESHOLD` для гібридного пошуку, якщо ви будете шукати не в масиві цілих чисел, а в масиві довгих рядків (де порівняння є дорожчою операцією)?
8. Тернарний пошук. Чи можна створити гібридний тернарно-лінійний пошук? Для яких задач він міг би бути корисним?
9. Есе. Напишіть есе на тему "Асимптотична складність проти реальної продуктивності: роль мікрооптимізацій та гібридних алгоритмів".
10. Зворотна задача. При якому значенні `THRESHOLD` гібридний пошук перетворюється на чистий лінійний пошук? А при якому — на чистий двійковий?
11. Робота з кешем. Поясніть, чому для малих `N` лінійний пошук може бути кеш-ефективнішим за двійковий.
12. Трирівневий гібрид. Запропонуйте ідею трирівневого гібридного пошуку (напр., інтерполяційний → двійковий → лінійний). В яких випадках він міг би бути доцільним?

13. Адаптивний поріг. Подумайте над ідеєю алгоритму, який би не мав фіксованого порогу, а адаптивно визначав його на основі характеристик вхідних даних.

14. Витрати на виклик функції. Порівняйте продуктивність гібридного алгоритму, де лінійний пошук винесений в окрему функцію, з версією, де код лінійного пошуку вбудований безпосередньо (inlined).

15. Двійковий пошук у зв'язаному списку. Чи має сенс реалізовувати будь-яку версію двійкового пошуку для зв'язаного списку? Обґрунтуйте, чому ні.

Пошук з режимом галопу

1. Теоретичні відомості

1.1. Ідея галопу: прискорення для структурованих даних

Уявіть, що ви сортуєте дві колоди карт, A і B, які вже відсортовані самі по собі, і вам потрібно злити їх в одну велику відсортовану колоду C. Ви берете першу карту з колоди B і шукаєте, куди її вставити в колоду A. Якщо виявляється, що перші 20 карт у колоді A значно менші за вашу карту з B, чи є сенс використовувати двійковий пошук, який почне перевіряти середину колоди A? Інтуїтивно, ні. Ми очікуємо, що місце для нашої карти знаходиться десь далі.

Пошук з режимом галопу, який є варіацією експоненційного пошуку, розв'язує саме цю задачу. Його головна мета — швидко пропустити великий блок елементів, які гарантовано не підходять, і знайти невеликий діапазон для більш точного пошуку. Цей метод особливо ефективний, коли дані мають певну структуру — наприклад, коли елементи з одного відсортованого масиву переважно більші (або менші) за елементи іншого.

Алгоритм працює у дві фази:

1. Крок 1: Галоп (Експоненційні стрибки). Замість того, щоб перевіряти елементи по одному, алгоритм "галопує" по масиву, експоненційно збільшуючи крок. Він перевіряє індекси 0, 1, 3, 7, 15, ..., $2^k - 1$. Цей процес триває, доки алгоритм не "перестрибне" позицію, де має знаходитись шуканий елемент. Тобто, доки `A[`$2^k - 1$`]` не стане більшим за `key`. Це дозволяє за логарифмічний час знайти діапазон `[2^{k-1}-1, 2^k-1]`, в якому міститься наш елемент.

2. Крок 2: Двійковий пошук. Після того, як ми визначили цей відносно невеликий діапазон, ми запускаємо в ньому звичайний двійковий пошук, щоб точно знайти потрібну позицію.

Цей підхід є адаптивним: якщо шуканий елемент знаходиться близько до початку, Крок галопу завершиться дуже швидко.

1.2. Псевдокод та його аналіз

Давайте формалізуємо цей процес. Функція буде шукати позицію для вставки елемента `key` у відсортований масив `A`.

```
// Вхід: A - відсортований масив, key - значення, для якого шукаємо п
озицію,
// start_pos - початкова позиція для пошуку.
// Вихід: індекс, куди можна вставити key, зберігаючи порядок.
ФУНКЦІЯ GallopingSearch(Array A, key, start_pos):
    n = довжина(A)

    // --- Крок  1: Галоп уперед ---
    // Визначаємо діапазон для пошуку, рухаючись експоненційними крок
ами.
    last_pos = start_pos
    step = 1
    current_pos = start_pos + 1 // 2^1 - 1

    ПОКИ current_pos < n І A[current_pos] < key:
        last_pos = current_pos
        step = step * 2
        current_pos = start_pos + step - 1
    КІНЕЦЬ ПОКИ

    // Обмежуємо верхню межу розміром масиву
    current_pos = min(current_pos, n - 1)

    // (Примітка: в реальних реалізаціях може бути і галоп назад,
    // якщо A[start_pos] > key, але ми розглядаємо базовий випадок)

    // --- Крок  2: Двійковий пошук у знайденому діапазоні ---
    // Тепер ми знаємо, що ключ знаходиться між last_pos та current_p
os.
    // Використовуємо двійковий пошук для знаходження точної позиції.
    ПОВЕРНУТИ BinarySearchForInsertion(A, key, last_pos, current_pos)
КІНЕЦЬ ФУНКЦІЇ
```

```
// Допоміжна функція двійкового пошуку, що повертає індекс для вставк
и
ФУНКЦІЯ BinarySearchForInsertion(Array A, key, left, right):
    // ... реалізація двійкового пошуку, що повертає індекс,
    // куди можна вставити key (наприклад, lower_bound) ...
КІНЕЦЬ ФУНКЦІЇ
```

1.3. Аналіз складності

Складність галопу, як і експоненційного пошуку, залежить від позиції шуканого елемента i.

– Часова складність:
    - Крок 1 (Галоп): Щоб знайти діапазон, що містить індекс i, потрібно виконати близько $\log_2 i$ стрибків.
    - Крок 2 (Двійковий пошук): Розмір знайденого діапазону також пропорційний i. Двійковий пошук у ньому займе $O(\log i)$.
    - Загальна складність: $O(\log i) + O(\log i) = O(\log i)$.

У найгіршому випадку, коли елемент знаходиться в кінці масиву ($i \approx n$), складність стає $O(\log n)$.

1.4. Адаптивність в TimSort

У реальному алгоритмі сортування TimSort режим галопу не вмикається одразу. Алгоритм починає з лінійного або двійкового пошуку. І лише якщо він помічає, що елементи з одного масиву систематично вставляються далеко в інший, він переходить у “режим галопу” (galloping mode). Якщо ж галоп виявляється неефективним (постійно “перестрибує” ціль і змушений повертатись назад), TimSort повертається до звичайного двійкового пошуку. Ця адаптивність дозволяє алгоритму бути ефективним на даних різної структури.

2. Область застосування

Пошук з режимом галопу не є самостійним алгоритмом пошуку, як лінійний чи двійковий. Це, скоріше, внутрішня техніка оптимізації, яка використовується всередині інших, більш складних алгоритмів.

– Алгоритми сортування злиттям: Його основне застосування — прискорення фази злиття (merge) у зовнішніх та адаптивних алгоритмах сортування, як-от TimSort.
– Об’єднання відсортованих списків: У будь-якій задачі, де потрібно ефективно об’єднати дві впорядковані послідовності.

– Пошук у структурах даних: Може використовуватися для швидкого пошуку у великих збалансованих деревах або списках з пропусками (skip lists).

3. Приклади застосування

– Python (`list.sort()`) та Java (`Arrays.sort()` для об'єктів): Обидві ці стандартні функції використовують TimSort, який, у свою чергу, активно використовує режим галопу для оптимізації злиття відсортованих "прогонів" (runs). Якщо ви колись помічали, що сортування частково відсортованого масиву в Python працює надзвичайно швидко, — це заслуга, в тому числі, і галопу.

– Системи управління базами даних: При злитті індексних блоків або відсортованих результатів запитів можуть застосовуватися схожі техніки.

Переваги та недоліки

Основна перевага галопу — його видатна продуктивність на даних, що мають "структуру збігу". Коли елементи з одного набору даних, ймовірно, знаходяться близько один до одного в іншому наборі, галоп дозволяє дуже швидко пропустити великі нерелевантні блоки. Він ефективно поєднує швидкість експоненційного пошуку для знаходження діапазону та точність двійкового пошуку для локалізації елемента.

Головний недолік — це накладні витрати. Алгоритм є складнішим за простий двійковий пошук. На повністю випадкових даних, де немає жодної структури, він може бути трохи повільнішим за двійковий пошук через складнішу логіку та перевірки для входу/виходу з режиму галопу. Саме тому в TimSort він не використовується постійно, а вмикається лише тоді, коли це вигідно.

Контрольні запитання

1. У чому полягає основна ідея пошуку з режимом галопу?
2. Поясніть дві основні фази роботи цього алгоритму.
3. Чому цей алгоритм називають "адаптивним", особливо в контексті TimSort?
4. Яка часова складність галопу, і від якого параметра вона переважно залежить?

5. Наведіть приклад набору даних, де пошук з режимом галопу буде значно ефективнішим за звичайний двійковий пошук при злитті двох масивів.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та аналіз пошуку з режимом галопу.

2. Мета

Практичне засвоєння логіки двофазного пошуку (експоненційні стрибки + двійковий пошук). Аналіз його продуктивності на різних типах даних.

3. Завдання

Створити програму, яка реалізує функцію пошуку з режимом галопу. Провести порівняльний аналіз кількості порівнянь, виконаних галопом та стандартним двійковим пошуком.

4. Опис

Програма повинна містити реалізацію функції `GallopingSearch`, яка всередині себе викликає функцію двійкового пошуку. Тестування має проводитись на двох типах масивів: одному, де елементи, що шукаються, знаходяться близько до початкової позиції, і другому, де вони знаходяться далеко.

5. Порядок виконання

1. Підготуйте інструменти: Візьміть вашу реалізацію двійкового пошуку з попередніх робіт.
2. Реалізуйте фазу галопу: Напишіть функцію `GallopingSearch`, яка спочатку виконує експоненційні стрибки для знаходження діапазону. Додайте виведення на екран, щоб бачити, як змінюються `last_pos` та `current_pos`.
3. Інтегруйте двійковий пошук: Після знаходження діапазону, викличте вашу функцію двійкового пошуку для цього діапазону.
4. Створіть тестові набори:
    - Набір 1 (ідеальний для галопу): Створіть відсортований масив `A` з 1000 елементів (напр., `{10, 20, 30, ...}`). Оберіть для пошуку ключ `key=15` (знаходиться на початку).

- Набір 2 (гірший для галопу): Використовуйте той самий масив `A`, але оберіть для пошуку ключ `key=995`.

5. Проведіть експеримент: Для обох наборів даних запустіть вашу функцію галопу та стандартний двійковий пошук. Підрахуйте та порівняйте загальну кількість порівнянь для кожного випадку.

6. Результат

Додаток, що реалізує пошук з режимом галопу, та аналіз, що демонструє, коли цей метод є ефективнішим за двійковий пошук.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми.
2. Результати вимірювань кількості порівнянь для обох тестових наборів та обох алгоритмів.
3. Бути готовим відповісти на питання:
    - Пояснити, чому для першого набору даних галоп виграв (або мав би виграти).
    - Чому для другого набору даних результати двійкового пошуку та галопу є дуже близькими?
    - Яка асимптотична складність вашої реалізації?

Тести (прості)

1. Пошук з режимом галопу — це варіація:

a) Лінійного пошуку

b) Пошуку стрибками

c) Експоненційного пошуку

d) Інтерполяційного пошуку

2. Яка головна мета першої фази (галопу) в алгоритмі?

a) Відсортувати масив.

b) Знайти точний індекс елемента.

c) Швидко знайти невеликий діапазон, де може знаходитись елемент.

d) Перевірити кожен елемент.

3. Який алгоритм викликається у другій фазі галопу?

a) Лінійний пошук.

b) Двійковий пошук.

c) Інший галоп з меншим кроком.

d) Пошук з поверненням.

4. Для якого завдання пошук з режимом галопу є ключовою оптимізацією?

a) Для розв’язання задачі комівояжера.

b) Для алгоритму сортування TimSort.

c) Для обчислення чисел Фібоначчі.

d) Для пошуку в хеш-таблиці.

5. Яка часова складність пошуку з режимом галопу в найгіршому випадку?

a) $O(n)$

b) $O(\sqrt{n})$

c) $O(\log n)$

d) $O(n \log n)$

Тести підвищеної складності

1. У контексті алгоритму TimSort, що змушує його перейти в режим галопу при злитті двох масивів A та B?

a) Коли розмір масивів перевищує певне порогове значення.

b) Коли виявляється, що кілька елементів з одного масиву (напр., B) поспіль вставляються в інший масив (A) без “втручання” елементів з A.

c) Коли дані в масивах розподілені нерівномірно.

d) Коли масиви не відсортовані.

2. Яка ключова відмінність між пошуком з режимом галопу та експоненційним пошуком?

a) Галоп використовує двійковий пошук у другій фазі, а експоненційний — лінійний.

b) Експоненційний пошук має складність $O(\log n)$, а галоп — $O(\log i)$.

c) Це практично синоніми, але термін “галоп” частіше використовується для опису цієї техніки як частини складнішого алгоритму (як TimSort), що може динамічно вмикати/вимикати цей режим.

d) Галоп працює на невпорядкованих даних, а експоненційний — ні.

3. При злитті двох масивів `A = [10, 20, 30, 40, 50]` та `B = [1, 2, 3]`, де буде найбільш ефективно застосувати галоп?

a) При пошуку місця для `1` з масиву B в масиві A.

b) При пошуку місця для `10` з масиву A в масиві B.

c) Ефективність буде однаковою в обох випадках.

d) Галоп тут неефективний, краще використати простий лінійний пошук.

4. Що станеться, якщо у першій фазі галопу (експоненційні стрибки) ми випадково "перестрибнемо" кінець масиву?

a) Алгоритм видасть помилку "вихід за межі масиву".

b) Коректна реалізація повинна обробляти цей випадок, встановлюючи верхню межу діапазону для двійкового пошуку рівною останньому індексу масиву.

c) Алгоритм почне пошук з початку.

d) Алгоритм автоматично перемкнеться на лінійний пошук.

5. Ідея експоненційних стрибків може бути узагальнена. Якби ми використовували кроки, що є степенями трійки (1, 3, 9, 27,...), якою була б асимптотична складність першої фази?

a) $O(\log_2 i)$

b) $O(\log_3 i)$

c) $O(3^i)$

d) $O(i^3)$

Відповіді на тести підвищеної складності

1. b) Алгоритм TimSort є адаптивним. Він переходить у режим галопу, коли виявляє структуру в даних. Ситуація, коли кілька елементів з одного відсортованого масиву йдуть підряд, є саме такою структурою. Це сигнал, що, ймовірно, і наступні елементи будуть йти підряд, тому вигідніше швидко "перестрибнути" великий блок іншого масиву, замість того,

2. c) Фундаментальна механіка обох алгоритмів — експоненційні стрибки з подальшим двійковим пошуком — ідентична. Відмінність полягає в контексті. "Експоненційний пошук" зазвичай розглядають як самостійний алгоритм для масивів невідомого розміру. "Пошук з режимом галопу" — це

назва цієї ж техніки, коли вона використовується як адаптивна оптимізація всередині іншого алгоритму (наприклад, сортування), який може вмикати цей режим, коли це вигідно, і вимикати, коли ні.

3. b) Ефективність галопу максимальна, коли він може швидко пропустити багато елементів. У цьому сценарії, шукаючи місце для `10` (з A) в масиві `B = [1, 2, 3]`, галоп зробить кілька швидких стрибків і одразу визначить, що `10` більше за всі елементи `B`. У зворотному випадку, при пошуку `1` (з B) в масиві `A`, галоп зупиниться на першому ж елементі (`A[0]=10`), не встигнувши продемонструвати свою перевагу.

4. b) Надійна реалізація алгоритму завжди перевіряє межі масиву. Якщо експоненційний стрибок виходить за межі, це означає, що шуканий елемент (якщо він існує) знаходиться між попередньою позицією стрибка та кінцем масиву. Тому друга Крок (двійковий пошук) буде коректно викликана для цього звуженого, але правильного діапазону.

5. b) Складність першої фази визначається тим, скільки кроків потрібно, щоб "досягти" індексу `i`. Якщо на кожному кроці `k` ми перевіряємо індекс 3^k, то ми шукаємо таке `k`, що $3^k \geq i$. Взявши логарифм за основою 3 від обох частин, ми отримаємо, що кількість кроків `k` пропорційна $\log_3 i$. Хоча асимптотично це теж саме, що $O(\log i)$, ця відповідь є більш точною для даної реалізації.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте пошук з режимом галопу згідно з методичними вказівками.
2. Галоп назад. Модифікуйте ваш алгоритм, щоб він міг виконувати "галоп назад", якщо `A[start_pos]` виявиться більшим за `key`.
3. Аналіз TimSort. Прочитайте детальну статтю про роботу алгоритму TimSort. Опишіть своїми словами, які критерії він використовує для входу та виходу з "режиму галопу".
4. Порівняння з експоненційним пошуком. Яка практична різниця між вашою реалізацією галопу та реалізацією експоненційного пошуку з попередніх розділів?
5. Пошук першого входження. Адаптуйте алгоритм для пошуку першого входження елемента в масив, де можуть бути дублікати.

6. Узагальнення кроку. Проведіть емпіричний аналіз: як зміниться кількість порівнянь, якщо крок у фазі галопу буде не $2^k - 1$, а, наприклад, $3^k - 1$.
7. Галоп у злитті. Реалізуйте функцію злиття двох відсортованих масивів, яка всередині використовує пошук з режимом галопу для визначення, скільки елементів можна скопіювати за один раз.
8. Нижня межа. Розробіть алгоритм на основі галопу для пошуку першого елемента, що є не меншим за `key` (аналог `lower_bound`).
9. Есе-аналіз. Напишіть есе на тему “Адаптивні алгоритми: чому здатність” підлаштовуватися” під дані є ключовою для сучасної продуктивності?“.
10. Комбінований пошук. Створіть функцію, яка аналізує перші кілька елементів масиву та, залежно від їх розподілу, автоматично обирає для пошуку двійковий, інтерполяційний або галоп-пошук.

Ефективний пошук: хешування та дерева

У попередніх розділах ми детально розглянули лінійні структури даних. Ми встановили, що, незважаючи на їхню простоту та корисність, вони мають фундаментальний недолік: повільний пошук. У невпорядкованій колекції, щоб знайти потрібний елемент, у найгіршому випадку доведеться переглянути всі записи. Це лінійний пошук зі складністю $O(n)$, і ця “тиранія лінійного часу” робить такі структури неефективними для роботи з великими обсягами даних.

Навіть якщо ми відсортуємо масив і застосуємо швидкий двійковий пошук зі складністю $O(\log n)$, ми заплатимо за це високою вартістю модифікації даних ($O(n)$ для вставки чи видалення).

Перед нами постає ключове питання: чи існують способи організації даних, які дозволяють виконувати і пошук, і вставку, і видалення значно швидше за лінійний час?

Відповідь — так. Історія комп’ютерних наук запропонувала два принципово різні, але однаково потужні підходи до розв’язання цієї проблеми. Ці два шляхи — хешування та збалансовані дерева пошуку — і є предметом нашого вивчення в цьому розділі.

Шлях перший: “Магічна картотека” — філософія хешування

Уявіть бібліотекаря, який, почувши назву книги, не шукає її в каталозі, а миттєво знає точний номер полиці. Хешування працює за схожим принципом. Замість того, щоб порівнювати шуканий елемент з іншими, ми намагаємося математично обчислити його місцезнаходження.

Спеціальна хеш-функція перетворює будь-який ключ (наприклад, ім'я користувача) на індекс у масиві. Це дозволяє виконувати пошук, вставку та видалення за середній час $O(1)$ — асимптотично найшвидший можливий результат. Однак за цю швидкість доводиться платити: хешування повністю руйнує природний порядок елементів.

Шлях другий: "Ієрархічні рішення" — філософія дерев пошуку

Цей підхід, навпаки, ставить на перше місце збереження порядку. Дані організовуються не в лінію, а в ієрархічну структуру, що нагадує дерево рішень.

Збалансовані дерева пошуку гарантують, що будь-який шлях від кореня до листка є дуже коротким. Це дозволяє виконувати пошук, вставку та видалення за гарантований час $O(\log n)$. Найголовніше — вони зберігають впорядкованість, що дає змогу ефективно виконувати такі операції, як пошук наступного за величиною елемента або вибірка даних у певному діапазоні.

У наступних розділах ми детально розберемо кожну з цих філософій. Ми зануримось у світ хеш-функцій та методів боротьби з колізіями. Потім ми вивчимо, як працюють самозбалансовані дерева і чому вони є основою сучасних баз даних. Розуміння цих двох підходів відкриє вам двері до розробки по-справжньому ефективних програмних систем.

Хеш-структури

У попередньому вступі ми розглянули два фундаментальні шляхи до подолання тиранії лінійного пошуку: шлях порядку (дерева) та шлях обчислення (хешування). Перший шлях, який ми дослідимо — це шлях хешування, що обіцяє нам майже миттєвий пошук зі складністю, близькою до $O(1)$.

Уявіть собі ідеальну шафу для зберігання речей. Ви даєте їй будь-який предмет — книгу, ключ, документ, — а вона миттєво каже вам точний номер шухляди, де цей предмет лежить. Жодних переглядів каталогів, жодних послідовних перевірок. Ця "магія" називається хешування.

Серцем цього підходу є хеш-функція — спеціальний математичний алгоритм, який виконує одне завдання: перетворює дані довільного розміру (наприклад, рядок “John Doe”) у число фіксованого розміру — хеш-код (наприклад, 42). Цей хеш-код ми використовуємо як індекс у звичайному масиві для зберігання наших даних. Таким чином, замість пошуку шляхом порівняння, ми намагаємося обчислити адресу даних напряму.

Звісно, у реальному світі магія рідко буває ідеальною. Що, як для двох різних предметів шафа вкаже на ту саму шухляду? Ця проблема називається колізією і є центральним викликом при проєктуванні основної хеш-структури — хеш-таблиці.

У цьому розділі ми:

- Дізнаємось, якими мають бути хороші хеш-функції.
- Розглянемо основні стратегії боротьби з колізіями.
- Проаналізуємо, чому хеш-таблиці є такими швидкими в середньому, але можуть деградувати в найгіршому випадку.
- Познайомимось із просунутими імовірнісними структурами, як-от фільтри Блума.

Хеш-таблиці:

1. Теоретичні відомості

1.1. Хеш-функція: серце хеш-таблиці

У вступі ми порівняли хешування з магічною картотекою, яка миттєво визначає місцезнаходження предмета. Інструментом, що творить цю магію, є хеш-функція.

Хеш-функція — це детермінований алгоритм, який приймає на вхід дані довільного розміру (ключ) і перетворює їх на ціле число фіксованого розміру, яке називається хеш-кодом або просто хешем.

```
hash_code = hash_function(key)
```

Цей хеш-код потім перетворюється на індекс у масиві, який є основою нашої структури — хеш-таблиці. Зазвичай це робиться за допомогою операції взяття за модулем:

```
index = hash_code % table_size
```

Якість усієї хеш-таблиці на 90% залежить від якості хеш-функції. Ідеальна хеш-функція повинна мати такі властивості:

1. Детермінізм. Для одного й того ж ключа хеш-функція завжди повинна повертати однаковий хеш-код. `hash("abc")` має бути однаковим і сьогодні, і завтра.
2. Ефективність обчислення. Хеш-функція має працювати дуже швидко. Немає сенсу в миттєвому доступі, якщо обчислення адреси займає більше часу, ніж сам пошук.
3. Рівномірний розподіл. Це найважливіша властивість. Хеш-функція має розподіляти ключі по масиву якомога рівномірніше. Ідеально, якщо для будь-якого набору ключів кожен індекс масиву має однакову ймовірність бути обраним. Це мінімізує кількість колізій.
4. Лавинний ефект. Навіть незначна зміна у вхідному ключі (наприклад, зміна одного символу) має призводити до кардинальної зміни хеш-коду. Це допомагає уникнути ситуації, коли схожі ключі (напр., “user1”, “user2”) потрапляють у сусідні комірки, що може призвести до кластеризації.

1.2. Проблема колізій: коли двоє приходять в одні двері

Навіть найкраща хеш-функція не може гарантувати унікальний індекс для кожного можливого ключа. Простір можливих ключів (наприклад, усі можливі рядки) є значно більшим за розмір нашої хеш-таблиці. Тому неминуче виникне ситуація, коли два різні ключі (`key1` ≠ `key2`) отримають однаковий індекс (`hash(key1) % size == hash(key2) % size`). Ця ситуація називається колізією.

Боротьба з колізіями — це центральна задача при проєктуванні хеш-таблиць. Існує два фундаментальні підходи до її розв’язання: закрите та відкрите адресування.

1.3. Методи вирішення колізій

1.3.1. Закрите адресування (Метод ланцюжків)

Ідея: Цей метод є найпростішим та найпоширенішим. Замість того, щоб зберігати в кожній комірці масиву лише один елемент, ми зберігаємо там вказівник на зв’язаний список (або іншу колекцію) елементів.

Як це працює:

1. Вставка: Обчислюємо індекс для нового елемента. Переходимо до зв’язаного списку за цим індексом і просто додаємо новий елемент на його початок.
2. Пошук: Обчислюємо індекс. Отримуємо зв’язаний список і виконуємо звичайний лінійний пошук вже всередині цього маленького списку.
3. Видалення: Обчислюємо індекс, знаходимо елемент у зв’язаному списку і видаляємо його.

Псевдокод операцій:

```
// table - це масив зв'язаних списків
// hash_function - наша хеш-функція
// table_size - розмір масиву

ФУНКЦІЯ Insert(table, key, value):
    index = hash_function(key) % table_size
    // Шукаємо, чи не існує вже елемент з таким ключем у ланцюжку
    ДЛЯ кожного елемента node в table[index]:
        ЯКЩО node.key == key ТО
            node.value = value // Оновлюємо значення
            ПОВЕРНУТИ
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
    // Якщо не знайшли, додаємо новий вузол на початок ланцюжка
    Додати_на_початок(table[index], Створити_Вузол(key, value))
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ Search(table, key):
    index = hash_function(key) % table_size
    ДЛЯ кожного елемента node в table[index]:
        ЯКЩО node.key == key ТО
            ПОВЕРНУТИ node.value
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
    ПОВЕРНУТИ NULL // Не знайдено
КІНЕЦЬ ФУНКЦІЇ
```

Продуктивність методу ланцюжків залежить від коефіцієнта заповнення $\alpha = n/m$, де n — кількість елементів, а m — розмір таблиці. Середня довжина ланцюжка дорівнює α.

1.3.2. Відкрите адресування

Ідея: Всі елементи зберігаються безпосередньо у самому масиві. Якщо при вставці ми натрапляємо на колізію (комірка вже зайнята), ми просто

шукаємо наступну вільну комірку за певним правилом. Цей процес пошуку вільної комірки називається пробуванням (probing).

4. Лінійне пробування

– Ідея: Якщо комірка `index` зайнята, перевіряємо `index + 1`, потім `index + 2`, і так далі, по колу.

– Проблема: Призводить до первинної кластеризації. Елементи починають збиватися у великі суцільні блоки. Це значно збільшує час пошуку, оскільки для знаходження вільного місця доводиться пробігати довгі ланцюжки зайнятих комірок.

4. Квадратичне пробування

– Ідея: Щоб уникнути первинної кластеризації, ми стрибаємо не на 1, а на $1^2, 2^2, 3^2, \ldots$ позицій. Якщо `index` зайнята, перевіряємо `index + 1`, потім `index + 4`, `index + 9`, і так далі.

– Проблема: Хоча це вирішує проблему первинної кластеризації, може виникати вторинна кластеризація: елементи з однаковим початковим хешем будуть слідувати однаковій послідовності проб.

6. Подвійне хешування

– Ідея: Це найефективніший метод відкритого адресування. Ми використовуємо дві різні хеш-функції: `h1(key)` для визначення початкової позиції, і `h2(key)` — для визначення кроку проби.

– `index_i = (h1(key) + i * h2(key)) % table_size`

– Оскільки крок проби тепер залежить від самого ключа, елементи з однаковим початковим хешем будуть мати різні послідовності проб, що ефективно вирішує проблеми кластеризації.

1.3.3. Сучасні методи відкритого адресування

– Cuckoo Hashing: Використовуються дві хеш-функції, що вказують на дві можливі позиції для кожного елемента. При вставці, якщо обидві позиції зайняті, новий елемент “викидає” один зі старих, який, у свою чергу, намагається зайняти свою альтернативну позицію, потенційно спричиняючи ланцюжок “викидань”. Забезпечує пошук за $O(1)$ у найгіршому випадку.

– Hopscotch Hashing: Елементи з однаковим хешем гарантовано знаходяться в невеликому “сусідстві” від своєї початкової позиції. Це

дозволяє обмежити пошук невеликим вікном, покращуючи кеш-ефективність.

- Robin Hood Hashing: Намагається “зрівняти” всіх у правах. При колізії, якщо новий елемент знаходиться “далі” від своєї ідеальної позиції, ніж елемент, що вже займає комірку, вони міняються місцями. Це зменшує максимальну довжину ланцюжка проб.

1.4. Досконале хешування

А що, якби можна було взагалі уникнути колізій? Досконале хешування — це техніка, яка дозволяє побудувати хеш-таблицю для статичного (незмінного) набору ключів, де пошук елемента гарантовано виконується за $O(1)$ у найгіршому випадку.

Ідея: Це дворівнева структура.

1. Рівень 1: Ми маємо звичайну хеш-таблицю, де колізії вирішуються методом ланцюжків.
2. Рівень 2: Для кожного ланцюжка (якщо в ньому більше одного елемента) ми будуємо свою власну, маленьку, досконалу хеш-таблицю другого рівня. Розмір цієї маленької таблиці обирається як квадрат від кількості елементів у ланцюжку, а хеш-функція підбирається випадково, доки не буде знайдена така, що не дає колізій.

2. Область застосування

- Словники (Dictionaries/Maps): Це основне застосування. У Python `dict`, в Java `HashMap`, в C++ `std::unordered_map` — все це реалізації хеш-таблиць.
- Множини (Sets): Аналогічно, `set` та `unordered_set`.
- Кешування: Швидке зберігання та отримання вже обчислених результатів.
- Індексація в базах даних: Хеш-індекси використовуються для швидкого доступу до записів за унікальним ключем.
- Символьні таблиці в компіляторах: Для зберігання інформації про змінні та функції.

3. Приклади застосування

- Підрахунок частоти слів: Ви можете ітерувати по тексту і для кожного слова збільшувати лічильник у хеш-таблиці, де ключ — це слово, а значення — кількість його повторень.

– Системи аутентифікації: Зберігання пар логін-пароль (точніше, логін та хеш пароля).
– Маршрутизатори: Зберігання таблиць маршрутизації для швидкого визначення, куди відправити наступний мережевий пакет.

Переваги та недоліки

Переваги:

– Видатна середня швидкість: Час виконання операцій пошуку, вставки та видалення в середньому складає $O(1)$, що є асимптотично найкращим можливим результатом.
– Гнучкість: Легко працюють з даними різного типу, доки для них можна визначити хорошу хеш-функцію.

Недоліки:

– Погана продуктивність у найгіршому випадку: При великій кількості колізій продуктивність може впасти до $O(n)$.
– Втрата порядку: Хеш-таблиці не зберігають жодного порядку елементів. Неможливо ефективно ітерувати по елементах у відсортованому порядку.
– Складність вибору хеш-функції: Ефективність структури критично залежить від якості хеш-функції.
– Вимоги до пам’яті: Для зменшення кількості колізій таблиця часто має бути більшою за кількість елементів, що зберігаються, що призводить до невикористаної пам’яті.

Контрольні запитання

1. Назвіть та поясніть три ключові властивості хорошої хеш-функції.
2. У чому полягає фундаментальна відмінність між методами закритого та відкритого адресування для вирішення колізій?
3. Поясніть проблему первинної кластеризації, що виникає при лінійному пробуванні.
4. Як подвійне хешування вирішує проблеми кластеризації?
5. Для якого типу набору даних можна побудувати досконалу хеш-таблицю і яка її головна перевага?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація хеш-таблиці з використанням методу ланцюжків.

2. Мета

Практичне засвоєння принципів роботи хеш-таблиць, включаючи реалізацію хеш-функції, обробку колізій та виконання основних операцій.

3. Завдання

Створити клас `HashTable`, що реалізує АТД “Словник” (пари ключ-значення) за допомогою методу ланцюжків. Ключем має бути рядок, значенням — ціле число.

4. Опис

Клас `HashTable` буде містити всередині масив, кожен елемент якого є вказівником на голову однозв’язного списку. Вузли списку будуть зберігати пару (ключ, значення).

5. Порядок виконання

1. Створіть структуру вузла: Опишіть структуру `Node` для зв’язного списку, що зберігає `key` (рядок) та `value` (ціле число).
2. Реалізуйте хеш-функцію: Напишіть просту поліноміальну хеш-функцію для рядків.
3. Створіть клас `HashTable`: Ініціалізуйте в конструкторі масив (напр., розміром 16) і заповніть його `NULL`.
4. Реалізуйте метод `insert(key, value)`:
    - Обчисліть хеш та індекс для ключа.
    - Перевірте, чи не існує вже елемент з таким ключем у ланцюжку за цим індексом. Якщо так, оновіть його значення.
    - Якщо ні, додайте новий вузол на початок ланцюжка.
5. Реалізуйте метод `search(key)`: Обчисліть індекс та виконайте лінійний пошук у відповідному ланцюжку.
6. Реалізуйте метод `remove(key)`: Обчисліть індекс, знайдіть вузол у ланцюжку та видаліть його.
7. Протестуйте: Створіть об’єкт, додайте декілька елементів, у тому числі ті, що викличуть колізію, знайдіть, оновіть та видаліть елементи.

6. Результат

Повністю функціональний клас, що реалізує хеш-таблицю.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією всіх необхідних класів та методів.
2. Продемонструвати роботу додатку: показати вставку, пошук, оновлення, видалення та, найголовніше, коректну обробку колізії.
3. Бути готовим відповісти на питання:
    - Як працює ваша хеш-функція?
    - Що таке коефіцієнт заповнення і як він впливає на продуктивність вашої таблиці?
    - Яка часова складність операцій у вашій реалізації в середньому та найгіршому випадку?

Тести

1. Основне призначення хеш-функції — це:

a) Відсортувати дані.

b) Перетворити ключ на індекс у масиві.

c) Зашифрувати дані.

d) Стиснути дані.

2. Ситуація, коли два різні ключі отримують однаковий хеш-код, називається:

a) Переповнення.

b) Колізія.

c) Кластеризація.

d) Помилка.

3. Який метод вирішення колізій зберігає елементи поза основним масивом таблиці?

a) Лінійне пробування.

b) Метод ланцюжків.

c) Квадратичне пробування.

d) Подвійне хешування.

4. Яка середня часова складність операції пошуку в добре реалізованій хеш-таблиці?

a) $O(1)$

b) $O(\log n)$

c) $O(n)$

d) $O(n^2)$

5. Який головний недолік хеш-таблиць порівняно зі збалансованими деревами?

a) Повільніша вставка.

b) Використовують більше пам’яті.

c) Не зберігають порядок елементів.

d) Складніші в реалізації.

Відповіді на тести 1 - b) , 2 - b) , 3 - b) , 4 - a) , 5 - c)

Тести підвищеної складності

1. Ви реалізували хеш-таблицю з відкритим адресуванням та лінійним пробуванням. Якщо коефіцієнт заповнення наближається до 1, як поводитиметься середня кількість проб при вставці нового елемента?

a) Залишиться константною.

b) Зростатиме логарифмічно.

c) Зростатиме лінійно.

d) Зростатиме квадратично.

2. Чому для подвійного хешування критично важливо, щоб значення `h2(key)` було взаємно простим з розміром таблиці `m`?

a) Щоб уникнути ділення на нуль.

b) Щоб гарантувати, що послідовність проб пройде по всіх комірках таблиці, перш ніж повторитися.

c) Щоб хеш-функція `h2` завжди повертала парне число.

d) Щоб зменшити кількість колізій на початковому етапі.

3. Який метод вирішення колізій є найбільш кеш-ефективним при успішному пошуку?

a) Метод ланцюжків.

b) Подвійне хешування.

c) Лінійне пробування.

d) Досконале хешування.

4. Використання хеш-таблиці для реалізації АТД “Множина” (Set) є ефективним, але для якої з наступних операцій вона буде абсолютно непридатною?

a) Перевірка, чи елемент **x** належить множині.

b) Додавання елемента **x** до множини.

c) Видалення елемента **x** з множини.

d) Пошук елемента, що є наступним за величиною після **x**.

5. У чому полягає основний принцип Cuckoo Hashing?

a) Використання трьох хеш-функцій для кращого розподілу.

b) Зберігання елементів, що колідують, у сусідніх комірках.

c) Кожен елемент має дві можливі “домівки”, і при вставці він може “виселити” інший елемент, змушуючи його шукати свою альтернативну домівку.

d) Використання одного великого зв’язного списку для всіх колізій.

6. Досконале хешування гарантує відсутність колізій та пошук за $O(1)$ у найгіршому випадку, але має суттєве обмеження:

a) Працює лише для цілих чисел.

b) Працює лише для статичних наборів ключів, які не змінюються.

c) Вимагає дуже багато пам’яті, $O(n^2)$.

d) Неможливе в реалізації на практиці.

7. У методі Robin Hood Hashing, коли виникає колізія при вставці нового елемента, що відбувається?

a) Новий елемент додається в кінець ланцюжка проб.

b) Новий елемент порівнюється з елементом, що вже займає комірку, за їхньою “відстанню” від ідеальної позиції. Елемент, що “бідніший” (далі від дому), займає комірку.

c) Використовується друга хеш-функція для знаходження наступної позиції.

d) Розмір таблиці подвоюється.

Відповіді на тести підвищеної складності

1. d) Через ефект первинної кластеризації, довжина ланцюжків проб зростає дуже швидко, коли таблиця майже заповнена. Теоретична середня кількість проб для вставки пропорційна $1/(1-\alpha)^2$, де α - коефіцієнт заповнення. При $\alpha \to 1$, цей вираз зростає квадратично.

2. b) Якщо крок `h2(key)` та розмір таблиці `m` мають спільний дільник, послідовність проб буде "зациклена" і відвідає лише частину комірок. Взаємна простота гарантує, що всі комірки будуть перевірені.

3. c) Лінійне пробування перевіряє сусідні комірки в пам'яті (`i, i+1, i+2,...`), що створює ідеальну просторову локальність. Це дозволяє кеш-пам'яті процесора працювати максимально ефективно, завантажуючи наперед потрібні дані.

4. d) Хеш-таблиці не підтримують жодного порядку елементів. Щоб знайти наступний за величиною елемент, необхідно перебрати всі N елементів, що є операцією $O(N)$. Для цього потрібні впорядковані структури, як-от збалансовані дерева.

5. c) Це основний принцип "хешування зозулі". Елемент, що вставляється, намагається зайняти одну зі своїх двох позицій. Якщо вони зайняті, він "викидає" одного з мешканців, який, у свою чергу, намагається переїхати до своєї другої "домівки".

6. b) Процес побудови досконалої хеш-таблиці, що включає підбір "вдалих" хеш-функцій, є відносно дорогим. Якщо набір ключів буде постійно змінюватися, всю структуру доведеться перебудовувати, що нівелює перевагу миттєвого пошуку.

7. b) Це "справедливий" підхід "Робін Гуда". Він мінімізує максимальну відстань проби, не дозволяючи жодному елементу відійти занадто далеко від своєї ідеальної позиції, що покращує продуктивність у найгіршому випадку.

Завдання для самостійної роботи

1. Проектування хеш-функції. Запропонуйте та опишіть алгоритм хеш-функції для об'єкта "Студент", який має поля: ім'я (рядок), прізвище (рядок), номер залікової книжки (ціле число).

2. Аналіз колізій. Візьміть просту хеш-функцію (напр., сума ASCII-кодів символів) та таблицю розміром 11. Вручну вставте в неї 10 слів та проаналізуйте кількість колізій.
3. Реалізація відкритого адресування. Модифікуйте вашу лабораторну роботу, реалізувавши хеш-таблицю з відкритим адресуванням та лінійним пробуванням.
4. Порівняння методів. Емпірично порівняйте продуктивність методів ланцюжків та лінійного пробування при різних коефіцієнтах заповнення.
5. Подвійне хешування. Реалізуйте подвійне хешування. Переконайтесь, що ваша друга хеш-функція ніколи не повертає 0.
6. Видалення у відкритій адресації. Дослідіть проблему видалення елемента з хеш-таблиці з відкритим адресуванням. Чому не можна просто очистити комірку? Які існують підходи до розв'язання (напр., використання "заглушок")?
7. Ідеальна хеш-функція. Поясніть, чому проста хеш-функція `h(key) = key % m` є поганим вибором, якщо розмір таблиці `m` є степенем двійки.
8. Криптографічні хеш-функції. Дослідіть, чим відрізняються криптографічні хеш-функції (SHA-256) від тих, що використовуються в хеш-таблицях. Чи можна використовувати SHA-256 для хеш-таблиці? Які будуть переваги та недоліки?
9. Есе-аналіз. Напишіть есе на тему "Компроміси у вирішенні колізій: порівняння методу ланцюжків та відкритого адресування".
10. Рехешування. Детально опишіть та реалізуйте механізм рехешування для вашої хеш-таблиці: коли коефіцієнт заповнення перевищує певне значення (напр., 0.75), розмір таблиці подвоюється, і всі елементи вставляються заново.

Імовірнісні фільтри

Фільтр Блума

1. Теоретичні відомості

1.1. Вступ: проблема "чи бачив я це раніше?"

Уявіть, що ви створюєте сервіс для скорочення URL-адрес. Коли користувач вводить довгу адресу, ви генеруєте коротку, але перед цим вам потрібно перевірити, чи не генерували ви таку ж адресу раніше. Або уявіть, що Google

хоче перевіряти, чи не є нова веб-сторінка шкідливою, порівнюючи її з гігантською базою даних мільярдів відомих шкідливих сайтів.

У всіх цих випадках ми стикаємося з однією і тією ж задачею: ефективно перевірити належність елемента до дуже великої множини.

Звичайні структури даних, як-от хеш-таблиці або дерева, є чудовими, але вимагають зберігання всіх елементів. Якщо множина містить мільярди елементів, така структура може займати гігабайти або навіть терабайти пам'яті, що робить її використання непрактичним.

Що, якби існувала структура, яка б дозволила нам пожертвувати невеликою часткою точності заради колосальної економії пам'яті? Така структура існує, і це — фільтр Блума.

1.2. Ідея фільтра Блума: компроміс з імовірністю

Фільтр Блума — це імовірнісна структура даних, яка дозволяє дуже швидко та компактно перевірити, чи може елемент належати до множини.

Його ключова особливість полягає в асиметричній точності:

- Якщо фільтр Блума каже: "Ні, цього елемента точно немає в множині", — то це 100% правда. Помилок першого роду (false negatives) не буває.
- Якщо фільтр Блума каже: "Так, цей елемент, *можливо*, є в множині", — то це може бути правдою, а може бути і хибно-позитивним спрацюванням (false positive).

Ця властивість робить його ідеальним "фільтром" або "охоронцем" для більш дорогих ресурсів. Замість того, щоб щоразу робити повільний запит до бази даних на диску, ми спершу запитуємо надзвичайно швидкий фільтр Блума в пам'яті. І лише якщо він каже "можливо", ми робимо повну перевірку.

1.3. Як це працює: біти та багато хеш-функцій

Фільтр Блума — це напрочуд проста структура. Вона складається з двох компонентів:

1. Бітовий масив: Великий масив з m бітів, початково заповнений нулями.
2. Набір k незалежних хеш-функцій: Це k різних хеш-функцій ($h_1, h_2, \ldots, h_k$), кожна з яких рівномірно відображає вхідний ключ на діапазон індексів `[0, m-1]`.

Операція `add(item)`: Додавання елемента до множини. Щоб додати елемент, ми виконуємо наступні кроки:

1. Обчислюємо *k* хеш-кодів для нашого елемента, використовуючи кожну з *k* хеш-функцій.
2. Кожен отриманий хеш-код ми використовуємо як індекс у бітовому масиві.
3. Встановлюємо біти за цими *k* індексами в одиницю.

Псевдокод для додавання:

```
ФУНКЦІЯ add(bloom_filter, item):
    ДЛЯ і ВІД 1 ДО k: // k - кількість хеш-функцій
        // Обчислюємо хеш та перетворюємо його на індекс
        index = hash_function_i(item) % m // m - розмір бітового маси
ву

        // Встановлюємо відповідний біт в 1
        bloom_filter.bit_array[index] = 1
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ФУНКЦІЇ
```

Операція `contains(item)`: Перевірка належності. Щоб перевірити, чи є елемент у множині, ми робимо схожі кроки:

1. Обчислюємо *k* хеш-кодів для елемента.
2. Перевіряємо біти за всіма *k* отриманими індексами.
3. Якщо ВСІ біти за цими індексами дорівнюють 1, ми кажемо, що елемент, можливо, є в множині.
4. Якщо хоча б ОДИН біт дорівнює 0, ми можемо з абсолютною впевненістю сказати, що елемента в множині точно немає. Адже якби він там був, всі його біти були б встановлені в 1 на етапі додавання.

Псевдокод для перевірки:

```
ФУНКЦІЯ contains(bloom_filter, item):
    ДЛЯ і ВІД 1 ДО k:
        index = hash_function_i(item) % m

        // Якщо хоча б один біт не встановлено, елемента точно немає
        ЯКЩО bloom_filter.bit_array[index] == 0 ТО
            ПОВЕРНУТИ false
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
```

```
    // Якщо всі біти встановлені, елемент, можливо, є
    ПОВЕРНУТИ true
КІНЕЦЬ ФУНКЦІЇ
```

Важливо: З фільтра Блума неможливо видалити елемент. Спроба встановити біт назад у 0 може “зламати” інформацію про інші елементи, які також використовували цей біт.

1.4. Імовірність хибно-позитивного спрацювання

Чому виникають хибно-позитивні спрацювання? Уявіть, ми додали до фільтра слова “apple” та “banana”. Потім ми перевіряємо наявність слова “cat”. Може статися так, що хеш-функції для “cat” випадково вкажуть на ті самі біти, які вже були встановлені в одиницю для слів “apple” та “banana”. У цьому випадку фільтр помилково скаже, що “cat”, можливо, є в множині.

Імовірність такого спрацювання `p` залежить від трьох параметрів:

- `m` — розмір бітового масиву.
- `n` — кількість елементів, доданих до фільтра.
- `k` — кількість хеш-функцій.

Формула для наближеної оцінки `p` є досить складною: $p \approx (1 - e^{-kn/m})^k$.

Ключовий висновок з неї: для заданої кількості елементів `n` та бажаної ймовірності помилки `p` можна математично обчислити оптимальний розмір масиву `m` та оптимальну кількість хеш-функцій `k`.

2. Область застосування

Фільтри Блума використовуються у розподілених системах та при роботі з великими даними, де економія пам’яті та швидкість є критичними, а невелика ймовірність помилки є прийнятною.

- Бази даних: Для швидкої перевірки існування запису перед дорогим доступом до диска (напр., Google BigTable, Apache Cassandra використовують їх, щоб уникнути пошуку в SSTable-файлах, які гарантовано не містять потрібного ключа).
- Мережева безпека: Маршрутизатори можуть використовувати фільтри Блума для блокування пакетів зі шкідливих IP-адрес, зберігаючи мільйони таких адрес у невеликому обсязі пам’яті.
- Веб-кешування: Щоб визначити, чи варто кешувати певний ресурс.
- Розподілені файлові системи.

3. Приклади застосування

– Google Chrome: Раніше використовував фільтр Блума для локальної перевірки URL-адрес на фішинг. Список шкідливих сайтів величезний, але його фільтр Блума займав мало місця. Якщо фільтр спрацьовував, браузер робив повний запит до серверів Google для остаточної перевірки.
– Medium (блог-платформа): Використовує фільтри Блума, щоб не рекомендувати користувачеві статті, які він уже прочитав.
– Криптовалюта Bitcoin: Використовується в механізмі SPV (Simplified Payment Verification) для швидкої перевірки, чи транзакція належить певному гаманцю, не завантажуючи весь блокчейн.

Переваги та недоліки

Переваги:

– Надзвичайна компактність: Фільтр Блума може представляти величезну множину, використовуючи значно менше пам'яті, ніж будь-яка інша структура даних, що зберігає самі елементи.
– Дуже швидкі операції: Операції `add` та `contains` є надзвичайно швидкими, оскільки вимагають лише кількох обчислень хеш-функцій та доступів до масиву. Їхня складність — $O(k)$, що є константою.
– Можливість об'єднання: Два фільтри Блума однакового розміру та з однаковими хеш-функціями можна об'єднати в один за допомогою побітової операції OR.

Недоліки:

– Хибно-позитивні спрацювання: Це фундаментальний компроміс. Імовірність помилки можна зменшити, збільшивши розмір фільтра, але повністю усунути її неможливо.
– Неможливість видалення: У класичній реалізації видалення елемента не підтримується. (Існують розширення, як-от Counting Bloom Filter, що вирішують цю проблему ціною збільшення пам'яті).
– Складність налаштування: Вибір оптимальних параметрів `m` та `k` вимагає апріорного знання про кількість елементів, які будуть зберігатися.

Контрольні запитання

1. У чому полягає основний компроміс, на якому побудований фільтр Блума?

2. Поясніть, чому у фільтрі Блума не може бути хибно-негативних спрацювань (false negatives).
3. Опишіть алгоритм додавання елемента до фільтра Блума.
4. Чому неможливо видалити елемент з класичного фільтра Блума?
5. Наведіть приклад системи, де невелика ймовірність хибно-позитивного спрацювання є прийнятною платою за швидкість та економію пам'яті.

6. Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та аналіз фільтра Блума.

2. Мета

Практичне засвоєння принципів роботи імовірнісних структур даних. Реалізація фільтра Блума та емпіричне дослідження його головної характеристики — ймовірності хибно-позитивних спрацювань.

3. Завдання

Створити клас `BloomFilter`, який надає два методи: `add(item)` та `contains(item)`. Провести експеримент для оцінки реальної частоти хибно-позитивних спрацювань.

4. Опис

Клас буде інкапсулювати бітовий масив та набір хеш-функцій. Оскільки написання багатьох хороших хеш-функцій є складним, можна використати трюк: згенерувати дві хороші хеш-функції `h1(x)` та `h2(x)`, а решту *k-2* функцій отримати їхньою лінійною комбінацією: `h_i(x) = h1(x) + i * h2(x)`.

5. Порядок виконання

1. Створіть клас `BloomFilter`: В конструкторі він має приймати розмір масиву `m` та кількість хеш-функцій `k`. Ініціалізуйте бітовий масив нулями.
2. Реалізуйте `add(item)`: Керуючись псевдокодом, реалізуйте логіку встановлення *k* бітів в одиницю.
3. Реалізуйте `contains(item)`: Реалізуйте логіку перевірки *k* бітів.
4. Проведіть експеримент:

- Створіть фільтр Блума з певними параметрами (наприклад, `m=1000`, `k=5`).
- Створіть дві множини слів: **`added_words`** (наприклад, 100 слів) та **`test_words`** (напр., 10000 слів, які гарантовано не перетинаються з першою множиною).
- Додайте всі слова з **`added_words`** до фільтра.
- Ітеруйте по **`test_words`**. Для кожного слова викликайте **`contains()`**. Підрахуйте, скільки разів функція поверне **`true`** (це і буде кількість хибно-позитивних спрацювань).

5. Обчисліть результат: Поділіть кількість хибно-позитивних спрацювань на загальну кількість тестових слів, щоб отримати реальну ймовірність помилки. Порівняйте її з теоретичною.
6. Результат

Додаток, що реалізує фільтр Блума, та звіт з емпіричною оцінкою його точності.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією фільтра Блума.
2. Результати експерименту: параметри фільтра, кількість доданих та перевірених елементів, отримана частота хибно-позитивних спрацювань.
3. Бути готовим відповісти на питання:
   - Як ви реалізували *k* хеш-функцій?
   - Як зміниться частота хибних спрацювань, якщо збільшити розмір масиву `m`? А якщо збільшити кількість хеш-функцій `k`?
   - Поясніть логіку роботи **`add`** та **`contains`** у вашому коді.

Тести

1. Фільтр Блума — це...

a) Структура даних для сортування.

b) Точна структура даних для перевірки належності.

c) Імовірнісна структура даних для перевірки належності.

d) Алгоритм шифрування.

2. Який тип помилки НЕМОЖЛИВИЙ у фільтрі Блума?

a) Хибно-позитивне спрацювання (False Positive).

b) Хибно-негативне спрацювання (False Negative).

c) Обидва типи можливі.

d) Жоден тип неможливий.

3. Яку операцію не підтримує класичний фільтр Блума?

a) Додавання елемента.

b) Перевірка належності елемента.

c) Видалення елемента.

d) Очищення фільтра.

4. Щоб додати елемент до фільтра Блума, потрібно:

a) Встановити один біт в 1.

b) Встановити *k* бітів в 1.

c) Встановити один біт в 0.

d) Записати елемент у масив.

5. Щоб зменшити ймовірність хибно-позитивного спрацювання, потрібно:

a) Зменшити розмір бітового масиву.

b) Збільшити розмір бітового масиву.

c) Використовувати лише одну хеш-функцію.

d) Зберігати елементи у зв'язному списку.

Відповіді на прості тести: 1-c, 2-b, 3-c, 4-b, 5-b

Тести підвищеної складності

1. Якщо два фільтри Блума, F1 та F2 (з однаковими `m` та `k`), об'єднати за допомогою побітової операції `OR` в новий фільтр F3, то `F3.contains(x)` буде `true` тоді і тільки тоді, коли:

a) `x` міститься і в F1, і в F2.

b) `x` міститься або в F1, або в F2 (або в обох).

c) `x` міститься тільки в F1.

d) `x` міститься тільки в F2.

2. Що станеться з імовірністю хибно-позитивних спрацювань, якщо для фіксованих `m` та `n` обрати занадто велику кількість хеш-функцій `k`?

a) Імовірність буде прямувати до 0, оскільки більше перевірок — вища точність.

b) Імовірність не зміниться, оскільки вона залежить лише від `m` та `n`.

c) Імовірність почне зростати і прямувати до 1, оскільки бітовий масив швидко заповниться одиницями.

d) Алгоритм перестане працювати через переповнення.

3. “Counting Bloom Filter” вирішує проблему неможливості видалення, замінюючи бітовий масив на масив лічильників. Яка нова проблема виникає при такому підході?

a) Алгоритм стає значно повільнішим.

b) Він починає допускати хибно-негативні спрацювання.

c) Виникає ризик переповнення лічильників, якщо один і той самий елемент додати багато разів.

d) Значно зростають вимоги до пам’яті (замість 1 біта на комірку потрібно, наприклад, 4 або 8 бітів).

4. У вас є фільтр Блума, розрахований на 1 мільйон елементів з імовірністю помилки 1%. Ви додали до нього 2 мільйони елементів. Що станеться з імовірністю хибно-позитивних спрацювань?

a) Залишиться 1%, оскільки параметри фільтра фіксовані.

b) Зменшиться, оскільки більше бітів буде встановлено.

c) Значно зросте, оскільки коефіцієнт заповнення став набагато вищим за оптимальний.

d) Неможливо передбачити.

Коротке пояснення: Параметри фільтра Блума (`m` та `k`) оптимізуються для конкретної очікуваної кількості елементів `n`. Якщо додати значно більше елементів, бітовий масив переповниться, і ймовірність колізій різко зросте. *Відповідь: c)*

5. Яка з операцій є фундаментально несумісною з принципом роботи фільтра Блума?

a) Об’єднання двох фільтрів (Union).

b) Перетин двох фільтрів (Intersection).

c) Підрахунок приблизної кількості елементів.

d) Перевірка належності.

6. Який зв’язок між фільтром Блума та хеш-таблицями?

a) Жодного, це абсолютно різні структури.

b) Фільтр Блума — це спрощена версія хеш-таблиці без вирішення колізій.

c) Обидві структури використовують хеш-функції для відображення ключів на індекси масиву, але фільтр Блума не зберігає самі ключі.

d) Хеш-таблиці використовуються всередині фільтра Блума.

7. Використання *k* незалежних хеш-функцій може бути обчислювально дорогим. Який трюк часто використовують на практиці, щоб згенерувати *k* хеш-значень, маючи лише дві хороші хеш-функції `h1(x)` та `h2(x)`?

a) Використовувати `h1(x)` для парних ітерацій та `h2(x)` для непарних.

b) Обчислювати `i`-тий хеш за формулою `h_i(x) = h1(x) + i * h2(x)`.

c) Розділити результат `h1(x)` на `k` частин.

d) Цього зробити неможливо, потрібно рівно *k* незалежних функцій.

Відповіді

1. b) Операція `OR` об’єднує множини. Якщо біт встановлено в будь-якому з операндів, він буде встановлений і в результаті. Таким чином, F3 буде ефективно представляти об’єднання множин, представлених F1 та F2.

2. *c)* Існує оптимальне значення `k`. Якщо `k` занадто велике, кожна операція `add` буде встановлювати забагато бітів. Дуже швидко майже весь масив заповниться одиницями, і будь-яка перевірка `contains` буде повертати `true`.

3. *d)* Це прямий наслідок зміни структури. Щоб зберігати лічильник, потрібно більше бітів на кожну “комірку”, ніж для простого прапорця 0/1. Проблема переповнення також існує, але вона є вторинною.

4. *c)* Параметри фільтра Блума (`m` та `k`) оптимізуються для конкретної очікуваної кількості елементів `n`. Якщо додати значно більше елементів, бітовий масив переповниться, і ймовірність колізій різко зросте.

5. *b)* Перетин двох фільтрів (побітове AND) може призвести до хибно-негативних спрацювань. Наприклад, якщо для елемента `x` у фільтрі F1 встановлені біти `{1, 5, 10}`, а у F2 — `{1, 6, 12}`, то `x` належить обом

множинам. Але їхній побітовий перетин дасть лише `{1}`, і при перевірці `contains(x)` по перетину ми побачимо, що біти 5, 6, 10, 12 не встановлені, і помилково вирішимо, що `x` не належить перетину.

6. *c)* Обидві структури базуються на одній і тій самій ідеї — хешуванні ключів для отримання індексів. Ключова відмінність у тому, що хеш-таблиця зберігає оригінальні ключі (щоб вирішувати колізії і повертати значення), а фільтр Блума зберігає лише "відбитки" ключів у вигляді встановлених бітів.

7. b) Ця техніка, відома як подвійне хешування або хешування Кірша-Мітценмахера, дозволяє симулювати *k* хеш-функцій, виконавши лише два основних обчислення. Це значно прискорює роботу фільтра, зберігаючи при цьому хороші статистичні властивості.

Завдання для самостійної роботи

1. Ручне трасування. Створіть бітовий масив з m=15 та 2 хеш-функції: `h1(x) = x % 15` та `h2(x) = (x*7) % 15`. Додайте до фільтра числа 10 та 20. Покажіть, які біти будуть встановлені. Перевірте, чи є в множині число 30.
2. Оптимальні параметри. Напишіть програму, яка для заданої кількості елементів `n` та бажаної ймовірності помилки `p` обчислює оптимальний розмір масиву `m` та кількість хеш-функцій `k`.
3. Порівняння з `HashSet`. Проведіть експеримент: створіть фільтр Блума та `HashSet` (або `std::unordered_set`). Додайте до них 1 мільйон випадкових рядків. Порівняйте обсяг пам'яті, який займає кожна структура.
4. Counting Bloom Filter. Реалізуйте Counting Bloom Filter, який використовує масив лічильників замість бітів, та додайте до нього функцію `delete(item)`.
5. Трюк з двома хеш-функціями. Модифікуйте вашу реалізацію фільтра Блума так, щоб вона використовувала техніку `h1(x) + i * h2(x)` для генерації *k* хешів.
6. Фільтр Блума для IP-адрес. Напишіть програму, що імітує мережевий фільтр. Створіть "чорний список" з 1000 IP-адрес, додайте їх до фільтра Блума. Потім перевірте 10000 випадкових IP-адрес, серед яких є 100 з чорного списку, та оцініть кількість хибно-позитивних та хибно-негативних спрацювань.

7. Імовірність на практиці. Проведіть експеримент: для фіксованих `m` та `n` змінюйте кількість хеш-функцій `k` від 1 до 20. Побудуйте графік залежності реальної частоти хибних спрацювань від `k`. Переконайтесь, що існує оптимальне значення `k`.
8. Об'єднання та перетин. Реалізуйте функції для об'єднання (union) та перетину (intersection) двох фільтрів Блума. Поясніть, чому перетин може призводити до хибно-негативних спрацювань.
9. Есе-аналіз. Напишіть есе на тему "Імовірнісні структури даних як інструмент для роботи з Big Data: коли точністю можна пожертвувати?".
10. Масштабований фільтр Блума (Scalable Bloom Filter). Дослідіть цю варіацію, яка дозволяє додавати елементи, навіть коли початкова місткість вичерпана. Опишіть, як вона працює.

Фільтр Блума, що рахує

Розглянувши класичний фільтр Блума, ми зіткнулися з його фундаментальним обмеженням — неможливістю видаляти елементи. Цей розділ присвячений елегантному розширенню цієї ідеї, яке вирішує цю проблему, хоча і вводить власні цікаві компроміси.

1. Теоретичні відомості

1.1. Проблема видалення та її джерело

У попередньому розділі ми встановили, що фільтр Блума є надзвичайно ефективним за пам'яттю та швидкістю для перевірки належності елемента до множини. Його робота базується на встановленні k бітів у бітовому масиві в одиницю для кожного доданого елемента.

Це призводить до ключового обмеження: неможливості видалення. Чому? Уявіть, що ми додали до фільтра два елементи, "apple" та "pear". Могло статися так, що одна з хеш-функцій для "apple" та одна з хеш-функцій для "pear" вказали на той самий індекс, скажімо, 42. Біт за цим індексом був встановлений в 1.

Тепер, якщо ми захочемо видалити "apple", нам потрібно буде "обнулити" всі його біти. Але якщо ми обнулимо біт за індексом 42, ми також пошкодимо інформацію про наявність елемента "pear", для якого цей біт також є важливим. Це може призвести до хибно-негативних спрацювань (false negatives), що є неприпустимим, оскільки руйнує основну гарантію фільтра Блума.

1.2. Ідея фільтра, що рахує: від бітів до лічильників

Як вирішити цю проблему? Ідея полягає в тому, щоб зберігати більше інформації в кожній комірці. Замість того, щоб просто знати, чи була комірка “зачеплена” хоч раз (стан 0/1), нам потрібно знати, скільки разів вона була зачеплена.

Фільтр Блума, що рахує (Counting Bloom Filter, CBF) — це варіація класичного фільтра Блума, яка замінює бітовий масив на масив невеликих лічильників.

Структура:

1. Масив лічильників (Counter Array): Масив з *m* комірок. Кожна комірка тепер зберігає не один біт, а невелике ціле число (наприклад, 4-бітний або 8-бітний лічильник).
2. Набір *k* незалежних хеш-функцій: Як і в класичній версії.

1.3. Операції над фільтром, що рахує

Логіка операцій залишається схожою, але замість встановлення бітів ми маніпулюємо лічильниками.

Операція `add(item)`: Інкрементація лічильників Щоб додати елемент, ми так само обчислюємо *k* хешів, але замість того, щоб встановлювати біти в 1, ми інкрементуємо (збільшуємо на 1) значення лічильників за відповідними індексами.

Псевдокод для додавання:

```
ФУНКЦІЯ add(counting_filter, item):
    ДЛЯ і ВІД 1 ДО k:
        index = hash_function_i(item) % m
        // Збільшуємо лічильник за цим індексом
        counting_filter.counter_array[index] = counting_filter.counte
r_array[index] + 1
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ФУНКЦІЇ
```

Операція `contains(item)`: Перевірка на ненульові значення Перевірка належності працює майже так само. Елемент вважається таким, що, *можливо*, належить до множини, якщо всі його лічильники більші за нуль.

Псевдокод для перевірки:

```
ФУНКЦІЯ contains(counting_filter, item):
    ДЛЯ і ВІД 1 ДО k:
        index = hash_function_i(item) % m
```

```
        // Якщо хоча б один лічильник нульовий, елемента точно немає
        ЯКЩО counting_filter.counter_array[index] == 0 ТО
            ПОВЕРНУТИ false
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    // Якщо всі лічильники більші за нуль, елемент, можливо, є
    ПОВЕРНУТИ true
КІНЕЦЬ ФУНКЦІЇ
```

Операція `delete(item)`: Декрементація лічильників Це ключова нова можливість. Щоб видалити елемент, ми декрементуємо (зменшуємо на 1) значення лічильників за його *k* індексами.

Псевдокод для видалення:

```
ФУНКЦІЯ delete(counting_filter, item):
    // Перед видаленням варто перевірити, чи елемент "можливо" є у фі
льтрі.
    // Видалення елемента, якого не було, може пошкодити дані.
    ЯКЩО contains(counting_filter, item) == false ТО
        // Нічого не робити або повідомити про помилку
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    ДЛЯ i ВІД 1 ДО k:
        index = hash_function_i(item) % m
        // Зменшуємо лічильник, але не нижче нуля
        ЯКЩО counting_filter.counter_array[index] > 0 ТО
             counting_filter.counter_array[index] = counting_filter.c
ounter_array[index] - 1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Нові виклики: розмір та переповнення

Можливість видалення дається не безкоштовно. Нова архітектура створює два нові виклики.

1. Вимоги до пам’яті. Це головний компроміс. Якщо класичний фільтр використовує 1 біт на комірку, то CBF, щоб зберігати лічильник, потребує більше. Зазвичай використовують 4-бітні лічильники (дозволяють рахувати до 15) або 8-бітні (до 255). Це означає, що Counting Bloom Filter займає в 4 або 8 разів більше пам’яті, ніж класичний фільтр Блума з таким самим розміром масиву `m`.

2. Переповнення лічильника. Що станеться, якщо ми додамо один і той самий елемент багато разів (або якщо різні елементи часто потрапляють в одну комірку) і лічильник, наприклад, 4-бітний, досягне свого максимального значення 15? Якщо ми спробуємо інкрементувати його ще раз, він може “обнулитися” (переповнитись). Це небезпечно.
    - Рішення: Зазвичай встановлюють правило, що лічильник, досягнувши максимального значення, більше не змінюється. Він назавжди залишається “максимальним”. Це означає, що будь-які елементи, пов’язані з цим лічильником, стають “безсмертними” — їх не можна буде коректно видалити. Однак це запобігає пошкодженню даних для інших елементів.

2. Область застосування

Counting Bloom Filter є ідеальним для динамічних множин, де елементи не тільки додаються, але й періодично видаляються.

- Мережевий аналіз: Відстеження активних мережевих потоків. Коли потік починається, він додається до фільтра, коли завершується — видаляється. Це дозволяє швидко перевіряти, чи є певний потік наразі активним.
- Кешування в базах даних: Бази даних можуть використовувати CBF для відстеження “гарячих” блоків даних, що знаходяться в кеші. Коли блок витісняється з кешу, він видаляється з фільтра.
- Виявлення унікальних елементів у потоці: Дозволяє підраховувати кількість унікальних елементів у потоці даних, де елементи можуть як з’являтися, так і зникати.

3. Приклади застосування

- Аналіз трафіку: Системи виявлення аномалій можуть використовувати CBF для зберігання “відбитків” нормальної мережевої активності. Якщо з’являється пакет, який не відповідає фільтру, він позначається як підозрілий. Патерни активності можуть змінюватися, тому можливість видаляти старі патерни є важливою.
- Рекомендаційні системи: Можна використовувати CBF для зберігання списку товарів, які користувач нещодавно переглянув, щоб не показувати їх у блоці “можливо, вас зацікавить”. Коли інтерес до товару зникає, його можна видалити з фільтра.

Переваги та недоліки

Переваги:

- Підтримка видалення: Це головна і єдина перевага над класичним фільтром Блума. Вона перетворює статичну структуру на динамічну.

Недоліки:

- Значно більше використання пам’яті: Це основний компроміс. За можливість видалення ми платимо в 4-16 разів більшим обсягом пам’яті.
- Ризик переповнення лічильників: Хоча існують стратегії боротьби, це додаткова складність, яка може призвести до того, що деякі елементи неможливо буде видалити.
- Складніша реалізація: Потребує роботи з масивом чисел, а не бітів, та обробки логіки інкременту/декременту.

Контрольні запитання

3. Поясніть, чому не можна безпечно видалити елемент з класичного фільтра Блума.
4. У чому полягає ключова відмінність у структурі між класичним та лічильниковим фільтром Блума?
5. Опишіть алгоритм видалення елемента з CBF.
6. Яка головна проблема, яку створює використання лічильників замість бітів?
7. Наведіть приклад задачі, де використання CBF є виправданим, а класичного фільтра Блума — недостатнім.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація Counting Bloom Filter та операції видалення.

2. Мета

Практичне засвоєння логіки роботи фільтра Блума, що рахує. Реалізація операцій додавання, перевірки та, найголовніше, видалення елемента.

3. Завдання

Створити клас **`CountingBloomFilter`**, який надає методи: **`add(item)`**, **`contains(item)`** та **`delete(item)`**. Продемонструвати коректність роботи всіх трьох операцій.

4. Опис

Клас буде інкапсулювати масив лічильників (наприклад, 8-бітних цілих чисел) та набір хеш-функцій.

5. Порядок виконання

1. Створіть клас **`CountingBloomFilter`**: В конструкторі він має приймати розмір масиву **`m`** та кількість хеш-функцій **`k`**. Ініціалізуйте масив лічильників нулями.
2. Реалізуйте **`add(item)`**: Керуючись псевдокодом, реалізуйте логіку інкрементації *k* лічильників. Додайте перевірку на переповнення (лічильник не повинен перевищувати свого максимального значення, напр., 255 для 8 біт).
3. Реалізуйте **`contains(item)`**: Реалізуйте логіку перевірки, що всі *k* лічильників більші за 0.
4. Реалізуйте **`delete(item)`**: Реалізуйте логіку декрементації *k* лічильників.
5. Протестуйте:
    - Додайте елемент “A”. Переконайтесь, що **`contains("A")`** повертає **`true`**.
    - Додайте елемент “B”.
    - Видаліть елемент “A”.
    - Переконайтесь, що **`contains("A")`** тепер повертає **`false`** (або **`true`** з певною ймовірністю, якщо були колізії), а **`contains("B")`** все ще повертає **`true`**.

6. Результат

Додаток, що демонструє коректну роботу всіх трьох операцій фільтра Блума, що рахує.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією класу **`CountingBloomFilter`**.
2. Продемонструвати роботу додатку, показавши сценарій додавання, видалення та перевірки елементів.
3. Бути готовим відповісти на питання:
    - Як ви обробляєте переповнення лічильників у вашому коді?

- Який обсяг пам’яті використовує ваша структура порівняно з класичним фільтром Блума з такими ж параметрами `m` та `k`?
- Поясніть логіку роботи методу `delete`.

Тести

1. Counting Bloom Filter відрізняється від класичного тим, що:

a) Використовує кращі хеш-функції.

b) Не допускає хибно-позитивних спрацювань.

c) Використовує масив лічильників замість бітового масиву.

d) Має логарифмічну складність.

2. Яку нову операцію дозволяє виконувати Counting Bloom Filter?

a) Додавання.

b) Пошук.

c) Видалення.

d) Сортування.

3. Який головний недолік Counting Bloom Filter?

a) Повільніші операції додавання та перевірки.

b) Значно більше використання пам’яті.

c) Неможливість об’єднання двох фільтрів.

d) Виникнення хибно-негативних спрацювань.

4. Щоб видалити елемент з Counting Bloom Filter, потрібно:

a) Встановити відповідні лічильники в 0.

b) Інкрементувати відповідні лічильники.

c) Декрементувати відповідні лічильники.

d) Перебудувати весь фільтр заново.

5. При перевірці належності елемента `contains(item)` у CBF, ми вважаємо, що елемент, можливо, є в множині, якщо:

a) Всі відповідні лічильники дорівнюють 1.

b) Хоча б один лічильник більший за 0.

c) Всі відповідні лічильники більші за 0.

d) Сума відповідних лічильників більша за k.

Відповіді на тести: 1-c, 2-c, 3-b, 4-c, 5-c

Тести підвищеної складності

1. Видалення елемента, якого ніколи не було у Counting Bloom Filter, може призвести до:

a) Помилки переповнення.

b) Появи хибно-негативних спрацювань для інших елементів.

c) Зменшення ймовірності хибно-позитивних спрацювань.

d) Жодних негативних наслідків, якщо лічильники не стають від'ємними.

2. Який компроміс виникає при виборі розрядності лічильників (напр., 4 біти проти 8 біт)?

a) 8-бітні лічильники працюють швидше, але займають більше місця.

b) 4-бітні лічильники займають менше місця, але мають значно вищий ризик переповнення, що може зробити видалення неможливим.

c) 4-бітні лічильники зменшують ймовірність хибно-позитивних спрацювань.

d) 8-бітні лічильники не потребують хеш-функцій.

3. Що станеться, якщо спробувати реалізувати перетин (Intersection) двох Counting Bloom Filters за допомогою покомпонентного мінімуму лічильників `C[i] = min(A[i], B[i])`?

a) Це коректний спосіб обчислити перетин.

b) Результат буде коректним, але операція буде дуже повільною.

c) Операція може призвести до хибно-негативних спрацювань.

d) Операція може призвести до хибно-позитивних спрацювань, але не до хибно-негативних.

4. У порівнянні з хеш-таблицею, яка дозволяє видалення, головна перевага Counting Bloom Filter полягає у:

a) Гарантованій відсутності колізій.

b) Значно меншому використанні пам'яті за рахунок відмови від зберігання самих ключів.

c) Часовій складності видалення $O(1)$ у найгіршому випадку.

d) Можливості зберігати елементи у впорядкованому вигляді.

5. Яка з наступних варіацій є найбільш підходящою для відстеження кількості унікальних відвідувачів веб-сайту в реальному часі?

a) Класичний фільтр Блума.

b) Counting Bloom Filter.

c) Структура HyperLogLog.

d) Збалансоване бінарне дерево.

6. Якщо в Counting Bloom Filter з 4-бітними лічильниками відбувається переповнення лічильника (він досягає 15), і ми приймаємо стратегію “не змінювати його більше”, що це означає для операції `delete`?

a) Операція `delete` буде працювати як завжди.

b) Операція `delete` для елемента, що використовує цей лічильник, призведе до помилки.

c) Декрементація цього лічильника буде ігноруватися, що потенційно унеможливлює видалення елемента.

d) Фільтр автоматично збільшить розрядність цього лічильника до 8 біт.

7. Ви хочете використовувати Counting Bloom Filter у системі, де деякі елементи є набагато важливішими за інші. Як можна модифікувати алгоритм, щоб важливі елементи було важче “випадково” видалити?

a) Це неможливо, всі елементи обробляються однаково.

b) Використовувати більше хеш-функцій (`k`) для важливих елементів, ніж для звичайних.

c) Збільшувати лічильники на 2 для важливих елементів, і на 1 — для звичайних.

d) Обидва варіанти b) та c) є валідними стратегіями.

Відповіді.

1. *b)* Якщо ми декрементуємо лічильники для елемента, якого не було, ми можемо помилково обнулити лічильник, який був встановлений іншим, коректним елементом. Це призведе до того, що при перевірці цього коректного елемента ми отримаємо `false`, тобто хибно-негативне спрацювання, що порушує основну гарантію фільтра.

2. *b)* Основний компроміс — це пам’ять проти надійності. Менші лічильники економлять пам’ять, але швидше “насичуються” і

переповнюються, після чого пов’язані з ними елементи не можна видалити. Більші лічильники надійніші, але дорожчі з точки зору пам’яті.

3. *c)* Як і в класичному фільтрі, перетин є проблематичним. Уявіть, елемент `x` належить обом множинам. У фільтрі A його лічильник `C1[i]` дорівнює 1. У фільтрі B цей же лічильник `C2[i]` дорівнює 2 (бо туди потрапив ще й елемент `y`). `min(1, 2) = 1`. Якщо ми потім видалимо `x` з обох множин, то в результаті `C[i]` стане 0, хоча елемент `y` все ще має бути присутнім у B. Це може призвести до хибно-негативних результатів.

4. *b)* Це фундаментальна перевага всіх фільтрів Блума. Вони не зберігають самі дані, а лише їхні “відбитки” у вигляді лічильників, що дозволяє досягти колосальної економії пам’яті, коли самі ключі є великими.

5. *c)* Це питання з підступом. Хоча CBF може відстежувати елементи, для задачі підрахунку кардинальності (кількості унікальних елементів) існують значно більш ефективні та спеціалізовані імовірнісні структури, як-от HyperLogLog, яка є стандартом де-факто для цієї задачі.

6. *c)* Якщо ми перестанемо зменшувати лічильник, він назавжди залишиться “встановленим”. Це означає, що будь-який елемент, який хешується в цю комірку, не може бути повністю видалений з фільтра (його “відбиток” частково залишиться), що є ціною за уникнення пошкодження даних для інших елементів.

7. *d)* Обидва підходи роблять “відбиток” важливого елемента більш вираженим. Більша кількість хешів (`k`) розподілить вплив по більшій кількості лічильників. Збільшення кроку інкременту зробить кожен окремий лічильник стійкішим до випадкових декрементацій від видалення інших елементів.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте Counting Bloom Filter згідно з методичними вказівками.
2. Аналіз переповнення. Проведіть експеримент: додайте один і той самий елемент до вашого CBF з 4-бітними лічильниками 20 разів. Перевірте, чи коректно обробляється переповнення (лічильники не повинні перевищувати 15).
3. Порівняння пам’яті. Створіть класичний фільтр Блума та CBF з однаковими `m` та `k`. Додайте до них 1000 елементів. Порівняйте обсяг пам’яті, який займає кожен з них.

4. Проблема видалення неіснуючого. Продемонструйте на прикладі, як видалення елемента, якого ніколи не було у фільтрі, може призвести до хибно-негативного спрацювання.
5. Есе-порівняння. Напишіть есе, що порівнює класичний фільтр Блума, Counting Bloom Filter та звичайну хеш-таблицю. Опишіть сценарії, де кожна з цих структур є найкращим вибором.
6. Оцінка кардинальності. Дослідіть, як можна використовувати CBF для приблизної оцінки кількості унікальних елементів у множині.
7. Оптимізація пам’яті. Запропонуйте ідею, як можна було б реалізувати CBF, використовуючи лічильники змінної довжини (напр., більшість лічильників 4-бітні, але деякі можуть “рости” до 8 біт, якщо це необхідно).

Фільтр Зозулі

Розглянувши фільтр Блума та його варіацію, що рахує, ми зіткнулися з їхніми фундаментальними компромісами: класичний фільтр не дозволяє видаляти елементи, а той, що рахує, розплачується за цю можливість значним збільшенням обсягу пам’яті.

Цей розділ присвячений більш сучасній та гнучкій альтернативі — фільтру Зозулі (Cuckoo Filter). Він не тільки вирішує проблему видалення, але й часто виявляється більш ефективним за пам’яттю та продуктивністю, ніж класичний фільтр Блума.

1. Теоретичні відомості

1.1. Ідея фільтра Зозулі: дві домівки та правило виселення

Уявіть собі невелике селище, де для кожного мешканця є рівно два будинки, в яких він може жити. Коли прибуває новий мешканець, він іде до свого першого будинку. Якщо той вільний — чудово, він заселяється. Якщо ж зайнятий, він іде до свого другого будинку. Якщо і той зайнятий, починається найцікавіше: новий мешканець поводиться як зозуля — він безцеремонно виселяє старого мешканця з одного з будинків і займає його місце. Виселений мешканець, у свою чергу, змушений іти до свого другого, альтернативного будинку, і якщо той теж зайнятий, він так само виселяє його мешканця. Цей ланцюжок “виселень” триває, доки хтось не знайде вільний будинок.

Ця аналогія лежить в основі хешування зозулі (Cuckoo Hashing), а фільтр Зозулі є його адаптацією для задач імовірнісної перевірки належності.

Фільтр Зозулі — це компактна імовірнісна структура даних, що дозволяє динамічно додавати та видаляти елементи, зберігаючи при цьому низьку ймовірність хибно-позитивних спрацювань.

1.2. Структура фільтра

Фільтр Зозулі складається з одного основного компонента:

– Таблиця (масив) “відбитків” (Fingerprints): Це масив, що складається з *m* комірок (buckets). Кожна комірка, у свою чергу, може зберігати не один, а декілька (зазвичай 4 або 8) відбитків.

Що таке “відбиток”? На відміну від фільтра Блума, який зберігає лише біти, фільтр Зозулі зберігає невеликі хеші самих елементів, які називаються відбитками (fingerprints). Відбиток — це, по суті, скорочена, часткова інформація про елемент.

Як це працює:

4. Дві хеш-функції: Ми використовуємо дві хеш-функції. Перша, `h1(item)`, обчислює індекс першої можливої комірки для елемента.
5. Відбиток: Ми обчислюємо відбиток елемента, зазвичай за допомогою іншої хеш-функції: `f = fingerprint(item)`.
6. Друга комірка: Індекс другої можливої комірки обчислюється не за допомогою другої хеш-функції, а хитріше — на основі індексу першої та відбитка елемента: `h2 = h1 ⊕ hash(f)`. Символ ⊕ означає побітову операцію XOR.

Ця властивість (`h2 = h1 ⊕ hash(f)`, з якої також випливає `h1 = h2 ⊕ hash(f)`) є ключовою: знаючи одну з можливих комірок елемента та його відбиток, ми завжди можемо обчислити іншу.

1.3. Операції над фільтром Зозулі

Операція `contains(item)`: Перевірка двох місць Це найпростіша і найшвидша операція.

7. Обчислюємо відбиток `f` для елемента `item`.
8. Обчислюємо індекси двох можливих комірок: `i1 = h1(item)` та `i2 = i1 ⊕ hash(f)`.
9. Перевіряємо вміст комірки `i1` та комірки `i2`. Якщо хоча б в одній з них знаходиться наш відбиток `f`, ми кажемо, що елемент, можливо, є у фільтрі.

10. Якщо відбитка **f** немає в жодній з двох комірок, ми можемо з абсолютною впевненістю сказати, що елемента точно немає.

Псевдокод для перевірки:

```
ФУНКЦІЯ contains(cuckoo_filter, item):
    f = fingerprint(item)
    i1 = hash1(item) % m
    i2 = (i1 ⊕ hash_of(f)) % m

    // Перевіряємо першу можливу комірку
    ЯКЩО bucket_contains(cuckoo_filter.table[i1], f) ТО
        ПОВЕРНУТИ true
    КІНЕЦЬ ЯКЩО

    // Перевіряємо другу можливу комірку
    ЯКЩО bucket_contains(cuckoo_filter.table[i2], f) ТО
        ПОВЕРНУТИ true
    КІНЕЦЬ ЯКЩО

    ПОВЕРНУТИ false
КІНЕЦЬ ФУНКЦІЇ
```

Операція **add(item)**: Спроба заселення та “викидання зозулі” Це найскладніша операція, яка демонструє всю ідею алгоритму.

1. Обчислюємо відбиток **f** та два можливі індекси **i1** та **i2**.
2. Перевіряємо, чи є вільне місце в комірці **i1**. Якщо так — вставляємо туди **f** і завершуємо.
3. Якщо в **i1** немає місця, перевіряємо комірку **i2**. Якщо там є місце — вставляємо **f** і завершуємо.
4. Початок “викидань”: Якщо обидві комірки заповнені, ми обираємо одну з них (зазвичай випадково, наприклад **i1**), виселяємо з неї один зі старих відбитків (назвемо його **old_f**), а на його місце ставимо наш новий відбиток **f**.
5. Тепер у нас в руках “бездомний” відбиток **old_f** та його поточна позиція **i1**. Ми обчислюємо для нього альтернативну позицію: **new_pos = i1 ⊕ hash_of(old_f)**.
6. Ми намагаємося вставити **old_f** у комірку **new_pos**. Якщо там є місце — чудово, процес завершено. Якщо ні — ми знову виселяємо когось і повторюємо крок 5.

7. Цей ланцюжок переміщень продовжується доти, доки не буде знайдено вільне місце або доки кількість переміщень не перевищить певний ліміт (`MaxKicks`). Якщо ліміт перевищено, вважається, що фільтр переповнений, і операція додавання зазнає невдачі.

Псевдокод для додавання:

```
ФУНКЦІЯ add(cuckoo_filter, item):
    f = fingerprint(item)
    i1 = hash1(item) % m
    i2 = (i1 ⊕ hash_of(f)) % m

    // Спроба 1: вставити в першу комірку
    ЯКЩО try_insert(cuckoo_filter.table[i1], f) ТО ПОВЕРНУТИ true КІН
ЕЦЬ ЯКЩО
    // Спроба 2: вставити в другу комірку
    ЯКЩО try_insert(cuckoo_filter.table[i2], f) ТО ПОВЕРНУТИ true КІН
ЕЦЬ ЯКЩО

    // Початок ланцюжка виселень
    current_index = випадково_обрати(i1, i2)

    ДЛЯ kick_count ВІД 0 ДО MaxKicks:
        // Випадково обираємо жертву в комірці і міняємо її на наш ві
дбиток
        victim_f = випадково_обрати_відбиток_з(cuckoo_filter.table[cu
rrent_index])
        замінити_в_комірці(cuckoo_filter.table[current_index], victim
_f, f)
        f = victim_f // Тепер наш "бездомний" - це виселена жертва

        // Обчислюємо альтернативну домівку для жертви
        current_index = (current_index ⊕ hash_of(f)) % m

        // Намагаємось вставити жертву в її нову домівку
        ЯКЩО try_insert(cuckoo_filter.table[current_index], f) ТО
            ПОВЕРНУТИ true // Успіх!
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    // Якщо цикл завершився, а вільне місце не знайдено
    ПОВЕРНУТИ false // Помилка: фільтр переповнений
КІНЕЦЬ ФУНКЦІЇ
```

Операція `delete(item)`: Просте видалення Завдяки зберіганню відбитків, видалення є простою та надійною операцією.

1. Обчислюємо відбиток **f** та два можливі індекси **i1** та **i2**.
2. Шукаємо відбиток **f** у комірці **i1**. Якщо знайдено — видаляємо його і завершуємо.
3. Якщо в **i1** не знайдено, шукаємо в комірці **i2**. Якщо знайдено — видаляємо.

Псевдокод для видалення:

```
ФУНКЦІЯ delete(cuckoo_filter, item):
    f = fingerprint(item)
    i1 = hash1(item) % m
    i2 = (i1 ⊕ hash_of(f)) % m

    ЯКЩО try_delete(cuckoo_filter.table[i1], f) ТО
        ПОВЕРНУТИ true
    КІНЕЦЬ ЯКЩО

    ЯКЩО try_delete(cuckoo_filter.table[i2], f) ТО
        ПОВЕРНУТИ true
    КІНЕЦЬ ЯКЩО

    ПОВЕРНУТИ false // Елемент не знайдено
КІНЕЦЬ ФУНКЦІЇ
```

2. Область застосування

Фільтр Зозулі є прямим конкурентом фільтра Блума, але з можливістю видалення. Він використовується у схожих сферах, особливо там, де множина елементів є динамічною.

– Мережеве обладнання: Для відстеження мережевих потоків, MAC-адрес у комутаторах, зберігання стану з'єднань.
– Бази даних та системи зберігання: Як і фільтр Блума, для швидкого відсікання запитів до неіснуючих даних. Можливість видалення є перевагою при роботі з даними, що оновлюються.
– Обчислення на льоту: У системах, що вимагають підрахунку унікальних елементів або перевірки належності у потоці даних, який постійно змінюється.

3. Приклади застосування

– Вбудовані мережеві пристрої: Комутатори та маршрутизатори часто мають дуже обмежену швидку пам'ять. Фільтр Зозулі дозволяє їм ефективно зберігати великі таблиці станів.

– Системи виявлення вторгнень (IDS): Можуть використовувати фільтр для зберігання “відбитків” підозрілих, але короткоживучих мережевих сесій.
– Програмно-конфігуровані мережі (SDN): Для відстеження правил потоків.

Переваги та недоліки

Переваги:

– Підтримка видалення: Це ключова перевага над класичним фільтром Блума.
– Вища ефективність по пам’яті: При низьких та середніх ймовірностях хибних спрацювань (напр., < 3%) фільтр Зозулі зазвичай займає менше місця, ніж фільтр Блума.
– Дуже швидкий пошук: Операція `contains` вимагає лише двох доступів до пам’яті.
– Краща локальність доступу: Доступ до двох комірок є більш передбачуваним для кешу процесора, ніж k випадкових доступів у фільтрі Блума.

Недоліки:

– Складніша вставка: Операція `add` може ініціювати ланцюжок переміщень і, в рідкісних випадках, завершитися невдачею, вимагаючи перебудови таблиці.
– Ризик переповнення: При високому коефіцієнті заповнення (зазвичай > 95%) ймовірність невдалої вставки різко зростає.
– Вимоги до хеш-функцій: Як і для будь-якої хеш-структури, якість хеш-функцій є критично важливою.

Контрольні запитання

1. У чому полягає основна ідея хешування зозулі, що дала назву фільтру?
2. Поясніть, як у фільтрі Зозулі обчислюється адреса другої можливої комірки для елемента. Чому цей метод є важливим?
3. Опишіть покроково, що відбувається під час операції `add`, якщо обидві потенційні комірки для нового елемента зайняті.
4. Чому у фільтрі Зозулі, на відміну від класичного фільтра Блума, можлива операція видалення?

5. Порівняйте операцію `contains` у фільтрі Блума та фільтрі Зозулі з точки зору кількості доступів до пам'яті.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та аналіз фільтра Зозулі.

2. Мета

Практичне засвоєння логіки роботи фільтра Зозулі, включаючи реалізацію механізму "виселення". Емпіричне дослідження залежності успішності вставки від коефіцієнта заповнення.

3. Завдання

Створити клас `CuckooFilter`, що реалізує операції `add`, `contains` та `delete`. Провести експеримент для дослідження, як зростає кількість "виселень" при заповненні фільтра.

4. Опис

Клас буде інкапсулювати таблицю, де кожна комірка є невеликим масивом для зберігання відбитків. Вам знадобиться реалізувати три хеш-функції: одну для індексу та дві для відбитка (або одну, яка буде скомбінована з індексом для отримання другого індексу).

5. Порядок виконання

1. Створіть клас `CuckooFilter`: В конструкторі він має приймати розмір таблиці `m` та розмір комірки `b`. Ініціалізуйте таблицю.
2. Реалізуйте `contains`: Це найпростіший метод, почніть з нього.
3. Реалізуйте `delete`: Реалізуйте пошук та видалення відбитка з однієї з двох можливих комірок.
4. Реалізуйте `add`: Це найскладніша частина. Реалізуйте основну логіку з ланцюжком виселень та лімітом `MaxKicks`.
5. Проведіть експеримент:
    - Створіть фільтр (напр., 1000 комірок по 4 відбитки).
    - У циклі додавайте до нього випадкові елементи, відстежуючи загальну кількість "виселень" та кількість невдалих вставок.
    - Побудуйте графік залежності середньої кількості виселень на одну вставку від коефіцієнта заповнення фільтра.

6. Результат

Додаток, що реалізує фільтр Зозулі, та графік, що демонструє його поведінку при заповненні.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією класу `CuckooFilter`.
2. Результати експерименту (графік або таблицю) та висновки щодо оптимального коефіцієнта заповнення.
3. Бути готовим відповісти на питання:
    - Як працює механізм “виселення” у вашому коді?
    - Що відбувається у вашій програмі, коли досягається `MaxKicks`?
    - Порівняйте вашу реалізацію з фільтром Блума: які переваги та недоліки ви бачите на практиці?

Тести

1. Фільтр Зозулі дозволяє, на відміну від класичного фільтра Блума:

a) Швидше виконувати операцію додавання.

b) Використовувати менше хеш-функцій.

c) Видаляти елементи.

d) Гарантувати відсутність хибно-позитивних спрацювань.

2. Якщо при вставці нового елемента обидві його потенційні комірки зайняті, що робить фільтр Зозулі?

a) Повертає помилку переповнення.

b) Замінює випадковий елемент у фільтрі на новий.

c) “Виселяє” один зі старих елементів, змушуючи його шукати свою альтернативну домівку.

d) Подвоює розмір таблиці.

3. Яка ключова властивість дозволяє, знаючи один можливий індекс **`i1`** та відбиток **`f`**, обчислити інший індекс **`i2`**?

a) **`i2 = (i1 + hash(f)) % m`**

b) **`i2 = (i1 * hash(f)) % m`**

c) **`i2 = (i1 ⊕ hash(f)) % m`** (де ⊕ — це XOR)

d) **`i2 = hash2(item) % m`**

4. Операція **`contains(item)`** у фільтрі Зозулі вимагає:

a) *k* доступів до пам’яті.

b) Одного доступу до пам’яті.

c) Двох доступів до пам’яті.

d) Перегляду всієї таблиці.

5. Що з переліченого є недоліком фільтра Зозулі?

a) Низька ефективність по пам’яті.

b) Повільний пошук.

c) Неможливість видалення.

d) Імовірність невдалої вставки, що вимагає перебудови фільтра.

Відповіді на тести: 1-c, 2-c, 3-c, 4-c, 5-d

Тести підвищеної складності

1. Чому зберігання “відбитків” (fingerprints) замість одного біта дозволяє реалізувати видалення?

a) Тому що відбиток містить повну інформацію про ключ.

b) Тому що видалення конкретного відбитка не впливає на інші відбитки в тій самій або інших комірках, на відміну від обнулення спільного біта.

c) Тому що відбиток дозволяє використовувати лише одну хеш-функцію.

d) Тому що відбитки є більшими за розміром, ніж біти.

2. Який з параметрів має найбільший вплив на зменшення ймовірності невдалої вставки (коли ланцюжок виселень зациклюється)?

a) Розмір відбитка (кількість біт).

b) Кількість хеш-функцій.

c) Кількість комірок у таблиці (`m`).

d) Розмір комірки (кількість слотів для відбитків, `b`).

3. Якщо дві комірки `i1` та `i2` для елемента `x` випадково збігаються (`i1 == i2`), як це вплине на роботу фільтра?

a) Це призведе до помилки, алгоритм не може обробити таку ситуацію.

b) Це ніяк не вплине на роботу, оскільки обидві хеш-функції просто вказують на ту саму комірку.

c) Це значно погіршує продуктивність, оскільки зменшує кількість варіантів для розміщення елемента і збільшує ймовірність довгого ланцюжка виселень.

d) Це покращує продуктивність, оскільки пошук потрібно робити лише в одній комірці.

4. Порівнюючи фільтр Блума та фільтр Зозулі при однаковій імовірності хибно-позитивних спрацювань (напр., 1%), яке твердження є правильним?

a) Фільтр Блума завжди буде займати значно менше місця.

b) Фільтр Зозулі зазвичай досягає високого заповнення (>95%) і може бути більш компактним.

c) Обидва займають однаковий обсяг пам’яті.

d) Фільтр Зозулі завжди займає більше місця через зберігання відбитків.

5. При реалізації `delete(item)` у фільтрі Зозулі, чому не можна просто знайти та видалити відбиток, не перевіряючи його наявність спочатку?

a) Це може призвести до хибно-негативного спрацювання, якщо видалити відбиток, що відповідає іншому елементу, який випадково має такий самий відбиток та хеш.

b) Операція `delete` без `contains` працює значно повільніше.

c) Це може пошкодити структуру хеш-функцій.

d) Такої проблеми не існує, можна видаляти без попередньої перевірки.

6. Який зв’язок між хешуванням зозулі (алгоритм для хеш-таблиць) та фільтром Зозулі?

a) Це два абсолютно різні алгоритми.

b) Фільтр Зозулі — це імовірнісна адаптація хешування зозулі, яка зберігає лише відбитки ключів замість пар ключ-значення.

c) Хешування зозулі використовує фільтр Зозулі всередині для вирішення колізій.

d) Обидва алгоритми гарантують відсутність колізій.

7. Уявіть фільтр Зозулі з комірками розміром 1. Чим він буде схожий на класичну хеш-таблицю з відкритим адресуванням?

a) Нічим, це абсолютно різні структури.

b) Він буде схожий на подвійне хешування, оскільки кожен елемент має дві можливі позиції.

c) Він буде схожий на хеш-таблицю з ланцюжками, де кожен ланцюжок має довжину 1.

d) Він буде схожий на лінійне пробування, оскільки при колізії елемент переміщується на сусідню позицію.

Відповіді на тести підвищеної складності:

1. b) Ключова відмінність: у фільтрі Блума один біт може бути "спільним" для багатьох елементів. Його обнулення пошкоджує інформацію про всіх них. У фільтрі Зозулі кожен елемент має свій унікальний (з високою ймовірністю) відбиток. Видалення одного відбитка не зачіпає інші.

2. d) Збільшення кількості слотів у комірці (b) дає кожному елементу більше "простору для маневру" при виселенні, що драматично зменшує довжину ланцюжків переміщень та ймовірність зациклення.

3. c) Це створює "вузьке місце". Якщо звичайна комірка має b слотів, а її альтернативна пара — ще b слотів, то в цій ситуації елемент та його потенційні "жертви" можуть переміщатися лише в межах однієї комірки з b слотами.

4. b) Дослідження показують, що при коефіцієнті заповнення ~95% фільтр Зозулі часто є більш компактним, ніж оптимально налаштований фільтр Блума для тієї ж імовірності помилки. Фільтр Блума стає ефективнішим лише при дуже низьких вимогах до ймовірності помилки.

5. a) Хоча ймовірність колізії відбитків мала, вона існує. Якщо ми видалимо відбиток, не переконавшись, що він належить саме нашому елементу, ми можемо випадково видалити "відбиток-двійник" іншого елемента. Це призведе до хибно-негативного спрацювання для того, іншого, елемента.

6. b) Фільтр Зозулі бере за основу ідею хешування зозулі (дві хеш-функції, виселення), але адаптує її для імовірнісної перевірки належності, замінюючи зберігання повних ключів та значень на зберігання компактних відбитків.

7. b) Хоча і не повністю, але ідея наявності двох можливих позицій для елемента, які обчислюються за допомогою хеш-функцій, є спільною рисою з подвійним хешуванням. На відміну від лінійного пробування, тут немає послідовного пошуку вільної комірки.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте фільтр Зозулі згідно з методичними вказівками.
2. Аналіз заповнення. Проведіть експеримент: для фільтра фіксованого розміру додавайте елементи, доки не станеться 10 невдалих вставок. Визначте, при якому коефіцієнті заповнення (`кількість_елементів / (кількість_комірок * розмір_комірки)`) це відбувається.
3. Порівняння з фільтром Блума. Емпірично порівняйте використання пам'яті фільтром Зозулі та фільтром Блума для досягнення однакової ймовірності хибно-позитивних спрацювань.
4. Вплив розміру комірки. Проведіть експеримент: як змінюється максимальний коефіцієнт заповнення при зміні розміру комірки з 2 до 4, а потім до 8?
5. Вплив розміру відбитка. Дослідіть, як розмір відбитка (в бітах) впливає на ймовірність хибно-позитивних спрацювань.
6. Часткові відбитки. Подумайте, як можна було б модифікувати відношення $\texttt{i2} = \texttt{i1} \oplus \texttt{hash(f)}$, якби ми зберігали не повний відбиток `f`, а лише його частину.
7. Виявлення циклів. Замість того, щоб обмежувати кількість виселень, модифікуйте алгоритм `add` так, щоб він виявляв цикли в ланцюжку переміщень.
8. Есе-порівняння. Напишіть есе, що порівнює фільтр Блума, Counting Bloom Filter та фільтр Зозулі за критеріями: швидкість, пам'ять, підтримка видалення, складність реалізації.
9. Евристики вибору жертви. У вашій реалізації жертва для виселення обирається випадково. Подумайте, чи існують інші, можливо, кращі евристики?
10. Відновлення після невдачі. Опишіть стратегію, як можна перебудувати фільтр Зозулі, якщо операція `add` завершилася невдачею.
11. Практична реалізація. Знайдіть та проаналізуйте реалізацію фільтра Зозулі у будь-якій відкритій бібліотеці (напр., Guava від Google).

12. Проблема асоціативності. Чи є операція додавання у фільтр Зозулі асоціативною? Тобто, чи завжди `(add(F, x); add(F, y))` дасть той самий кінцевий стан, що й `(add(F, y); add(F, x))`?

Фільтр-частка

Ми підійшли до фінального розділу, присвяченого імовірнісним структурам даних. Ми побачили, як фільтр Блума досягає компактності ціною відмови від видалення, і як фільтр Зозулі повертає можливість видалення, але вводить складну логіку "виселень". Тепер ми розглянемо структуру, яка прагне взяти найкраще з обох світів: високу ефективність використання пам'яті, підтримку видалення та, найголовніше, чудову кеш-локальність, що робить її надзвичайно швидкою на практиці.

1. Теоретичні відомості

1.1. Ідея фільтра-частки: розумне кодування позиції

Уявіть, що вам потрібно організувати зберігання великої кількості пронумерованих папок. Замість того, щоб просто зберігати повний номер кожної папки, ви вирішуєте використати хитру систему. Ви ділите номер папки, наприклад 1342, на дві частини: "номер стелажа" (перші дві цифри, 13) та "номер полиці" (останні дві цифри, 42). Ви йдете до стелажа №13 і ставите туди картку, на якій написано лише "42".

Коли вам потрібно знайти папку 1342, ви знову ділите номер, йдете до стелажа 13 і шукаєте серед карток ту, на якій написано "42". Ви змогли зберегти інформацію, не записуючи повний номер. Частина інформації (номер стелажа) була закодована в самій позиції картки.

Фільтр-частка (Quotient Filter) — це імовірнісна структура даних, що працює за схожим принципом. Він використовує одну велику хеш-таблицю, але замість зберігання повного хешу елемента, він ділить його на дві частини:

- Частка (Quotient): Старші біти хешу, які використовуються як індекс у таблиці, вказуючи на "домашню" комірку елемента.
- Залишок (Remainder): Молодші біти хешу, які зберігаються у таблиці як "відбиток" елемента.

Ключова ідея полягає в тому, що нам не потрібно зберігати частку, оскільки ми можемо відновити її, знаючи позицію залишку в таблиці. Це дозволяє досягти надзвичайної компактності.

1.2. Структура фільтра та логіка "пробіжок"

Фільтр-частка — це, по суті, хеш-таблиця з відкритим адресуванням та дуже специфічною логікою вирішення колізій.

1. Хеш-функція: Ми використовуємо одну хеш-функцію `h(item)`, яка генерує p-бітний хеш.
2. Розбиття хешу: Цей хеш ділиться на дві частини:
    - q старших бітів — це частка, $h_q(item)$.
    - r молодших бітів — це залишок, $h_r(item)$. ($p = q + r$).
3. Таблиця: Ми маємо масив з $m = 2^q$ комірок. Кожна комірка може зберігати один r-бітний залишок.

Що відбувається при колізії? Якщо домашня комірка `h_q(item)` вже зайнята, ми використовуємо варіацію лінійного пробування. Ми просто шукаємо наступну вільну комірку, рухаючись вправо (і зациклюючись, якщо потрібно).

Але як тепер знайти наші елементи? Якщо залишки розкидані по таблиці, як ми дізнаємось, який залишок до якої частки (домашньої комірки) належить? Для цього фільтр-частка використовує геніальну систему метаданих. Кожна комірка, окрім залишку, зберігає три додаткові біти:

– `is_occupied`: показує, чи є ця комірка початком “пробіжки” для якогось хешу.

– `is_continuation`: показує, чи є цей елемент продовженням пробіжки попереднього.

– `is_shifted`: показує, чи знаходиться цей елемент не у своїй “домашній” комірці, а був зсунутий через колізію.

Всі залишки, що належать до однієї частки `q`, утворюють неперервну “пробіжку” (run). Всі елементи з однаковою часткою гарантовано знаходяться в одному нерозривному блоці.

1.3. Операції над фільтром-часткою

Операція `add(item)`: Вставка зі зсувом

1. Обчислюємо хеш `h(item)` і ділимо його на частку `q` та залишок `r`.
2. Переходимо до домашньої комірки з індексом `q`.
3. Скануємо таблицю вправо, щоб знайти кінець пробіжки для частки `q`. Це робиться за допомогою метаданих.

4. Знайшовши місце для вставки (зберігаючи сортування залишків всередині пробіжки), ми вставляємо наш залишок `r`.
5. Всі наступні елементи (до першої порожньої комірки) зсуваються на одну позицію вправо.
6. Оновлюємо метадані для вставленого та зсунутих елементів.

Псевдокод для вставки (спрощена логіка):

```
ФУНКЦІЯ add(quotient_filter, item):
    hash = hash_function(item)
    q = get_quotient(hash)
    r = get_remainder(hash)

    // Якщо комірка q порожня і не є частиною іншої пробіжки, просто вставляємо
    ЯКЩО is_empty(quotient_filter.table[q]) ТО
        quotient_filter.table[q].remainder = r
        quotient_filter.table[q].is_occupied = true
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // Знаходимо початок кластера (першу порожню комірку зліва)
    // ... логіка пошуку ...

    // Знаходимо позицію для вставки всередині пробіжки q
    // ... логіка сканування пробіжки ...

    // Зсуваємо всі елементи від позиції вставки до кінця кластера вправо
    // ... логіка зсуву ...

    // Вставляємо наш залишок r на звільнене місце
    // ... логіка вставки ...

    // Оновлюємо метадані
    // ... is_occupied, is_continuation, is_shifted ...
КІНЕЦЬ ФУНКЦІЇ
```

Операція `contains(item)`: Пошук у пробіжці

7. Обчислюємо хеш `h(item)` і ділимо його на частку `q` та залишок `r`.
8. Переходимо до домашньої комірки `q`.
9. Якщо її біт `is_occupied` дорівнює 0, елемента точно немає.

10. Якщо `is_occupied` дорівнює 1, ми починаємо сканувати пробіжку для частки `q` вправо.
11. Якщо під час сканування ми знаходимо наш залишок `r`, елемент, можливо, є у фільтрі.
12. Якщо ми доходимо до кінця пробіжки і не знаходимо `r`, елемента точно немає.

Псевдокод для перевірки:

```
ФУНКЦІЯ contains(quotient_filter, item):
    hash = hash_function(item)
    q = get_quotient(hash)
    r = get_remainder(hash)

    // Якщо домашня комірка не є початком пробіжки, елемента немає
    ЯКЩО quotient_filter.table[q].is_occupied == false ТО
        ПОВЕРНУТИ false
    КІНЕЦЬ ЯКЩО

    // Скануємо пробіжку, що починається з комірки q
    // ... логіка сканування з використанням метаданих ...

    // Під час сканування шукаємо наш залишок r
    ЯКЩО знайдено_залишок(r) В пробіжці ТО
        ПОВЕРНУТИ true
    ІНАКШЕ
        ПОВЕРНУТИ false
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Операція `delete(item)` Видалення можливе і є відносно ефективним. Воно включає пошук елемента і подальший зсув решти елементів у кластері вліво, щоб заповнити порожнє місце.

2. Область застосування

Фільтр-частка є прямим конкурентом фільтрів Блума та Зозулі, пропонуючи унікальний набір характеристик.

- Вбудовані системи та мережеве обладнання: Де кеш-ефективність є абсолютно критичною.
- Системи баз даних та зберігання: Особливо в LSM-деревах, де потрібно ефективно перевіряти наявність ключів.

– Біоінформатика: Для зберігання та аналізу великих наборів k-мерів (коротких послідовностей ДНК).

3. Приклади застосування

– RocksDB: Високопродуктивна вбудована база даних від Facebook використовує варіації фільтра-частки для оптимізації доступу до даних на SSD.

– Високошвидкісні маршрутизатори: Можуть використовувати фільтр-частку для відстеження великої кількості мережевих потоків при обмеженій пам'яті.

Переваги та недоліки

Переваги:

– Відмінна кеш-локальність: Завдяки використанню лінійного пробування, доступ до даних відбувається послідовно, що є ідеальним для кешів процесора. Це робить фільтр-частку значно швидшим на практиці, ніж фільтр Зозулі або Блума.

– Висока ефективність по пам'яті: Часто виявляється більш компактним, ніж фільтр Блума, особливо при низьких вимогах до ймовірності помилки.

– Підтримка видалення та динамічного розширення.

– Можливість об'єднання та зміни розміру "на льоту".

Недоліки:

– Найвища складність реалізації: З усіх імовірнісних фільтрів, фільтр-частка є найскладнішим для коректної реалізації через складну логіку роботи з метаданими та зсувами.

– Продуктивність деградує при високому заповненні: Як і будь-яка структура з відкритим адресуванням, при заповненні близькому до 100%, довжина пробіжок зростає, що сповільнює операції.

Контрольні запитання

1. У чому полягає основна ідея фільтра-частки, що відрізняє його від інших фільтрів?
2. Поясніть, як хеш елемента ділиться на частку та залишок, і яка роль кожної з цих частин.
3. Що таке "пробіжка" (run) у фільтрі-частці і як система дізнається, де вона починається і закінчується?

4. Чому фільтр-частка вважається більш кеш-ефективним, ніж фільтр Зозулі?
5. Опишіть процес вставки елемента, коли його домашня комірка вже зайнята.

Методичні вказівки до лабораторної роботи

1. Назва

Дослідження та реалізація спрощеного фільтра-частки.

2. Мета

Набути базового розуміння принципів роботи фільтра-частки шляхом реалізації його основних операцій (вставка, пошук) з використанням лінійного пробування.

3. Завдання

Створити клас **`QuotientFilter`**, що реалізує операції **`add`** та **`contains`**. Для спрощення, можна опустити роботу з бітовими метаданими, ідентифікуючи пробіжки іншим способом.

4. Опис

Клас буде інкапсулювати масив для зберігання залишків. Вам знадобиться реалізувати хеш-функцію та логіку її розбиття на частку та залишок, а також механізм лінійного пробування для вирішення колізій.

5. Порядок виконання

1. Створіть клас **`QuotientFilter`**: Визначте параметри **`q`** та **`r`**, ініціалізуйте таблицю розміром 2^q.
2. Реалізуйте хешування: Напишіть функцію, що приймає ключ і повертає його частку та залишок.
3. Реалізуйте **`add(item)`**:
    - Обчисліть **`q`** та **`r`**.
    - Знайдіть першу вільну комірку, починаючи з індексу **`q`** (лінійне пробування).
    - Вставте залишок **`r`** у знайдену комірку.
4. Реалізуйте **`contains(item)`**:
    - Обчисліть **`q`** та **`r`**.

- Починаючи з індексу `q`, скануйте таблицю, доки не зустрінете порожню комірку.
- Якщо під час сканування ви знайшли залишок `r`, поверніть `true`.

5. Протестуйте: Додайте декілька елементів, що викликають колізії, і перевірте, що пошук знаходить їх у правильних, зсунутих позиціях.

6. Результат

Додаток, що демонструє базові принципи роботи фільтра-частки.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми.
2. Продемонструвати роботу на прикладі з колізіями.
3. Бути готовим відповісти на питання:
   - Як ваша реалізація вирішує колізії?
   - Чому ця спрощена реалізація є менш ефективною, ніж повна версія з метаданими?
   - Яка складність операцій у вашій реалізації?

Тести

1. Фільтр-частка зберігає у своїх комірках:

a) Повні хеші елементів.

b) Лише молодші біти хешу (залишки).

c) Лише старші біти хешу (частки).

d) Самі елементи.

2. Який метод вирішення колізій використовує фільтр-частка?

a) Метод ланцюжків.

b) Варіацію лінійного пробування.

c) Квадратичне пробування.

d) Подвійне хешування.

3. Головна перевага фільтра-частки над фільтром Зозулі та Блума полягає у:

a) Меншій ймовірності хибно-позитивних спрацювань.

b) Простішій реалізації.

c) Вищій кеш-ефективності завдяки послідовному доступу до пам’яті.

d) Відсутності потреби в хеш-функціях.

4. Інформація про те, до якої “домашньої” комірки належить залишок, у фільтрі-частці:

a) Зберігається разом із залишком.

b) Не зберігається, її неможливо відновити.

c) Кодується в позиції залишку та спеціальних метаданих.

d) Обчислюється за допомогою другої хеш-функції.

5. Яка з операцій є найскладнішою для реалізації у фільтрі-частці?

a) Пошук (`contains`)

b) Вставка (`add`)

c) Видалення (`delete`)

d) Очищення фільтра

Відповіді на тести: 1-b, 2-b, 3-c, 4-c, 5-b

Тести підвищеної складності

1. У фільтрі-частці елементи з однаковою часткою `q` утворюють “пробіжку”. Як система знаходить кінець однієї пробіжки і початок наступної?

a) Кожен елемент зберігає вказівник на наступний.

b) Розмір кожної пробіжки зберігається в домашній комірці.

c) За допомогою трьох біт метаданих (`is_occupied`, `is_continuation`, `is_shifted`), які дозволяють відновити межі пробіжок.

d) Всі пробіжки мають фіксований розмір.

2. Порівнюючи фільтр-частку та хеш-таблицю з лінійним пробуванням, яка ключова відмінність у їхній поведінці при пошуку?

a) Хеш-таблиця не має колізій, а фільтр-частка має.

b) У хеш-таблиці потрібно порівнювати повні ключі, тоді як у фільтрі-частці пошук зупиняється, як тільки знайдено кінець пробіжки, навіть не досягнувши порожньої комірки.

c) Фільтр-частка завжди швидший.

d) У фільтрі-частці неможливе видалення, а в хеш-таблиці — можливе.

3. Чому фільтр-частка є особливо привабливим для LSM-дерев (Log-Structured Merge-Tree)?

a) Тому що він може зберігати дуже великі елементи.

b) Тому що він підтримує ефективне об’єднання (merge) двох фільтрів, що є частою операцією при компактифікації SSTable-файлів.

c) Тому що він не має хибно-позитивних спрацювань.

d) Тому що він не використовує хеш-функції.

4. Як впливає збільшення розміру залишку **`r`** (при фіксованому загальному розмірі хешу **`p=q+r`**) на характеристики фільтра-частки?

a) Збільшує кількість комірок у таблиці, зменшуючи ймовірність колізій.

b) Зменшує кількість комірок у таблиці, але зменшує ймовірність хибно-позитивних спрацювань, оскільки “відбитки” стають більш унікальними.

c) Ніяк не впливає на характеристики.

d) Покращує кеш-ефективність.

5. У чому полягає основний компроміс при виборі між фільтром Зозулі та фільтром-часткою?

a) Фільтр Зозулі швидший для вставки, але повільніший для пошуку.

b) Фільтр Зозулі використовує більше пам’яті, але простіший в реалізації.

c) Фільтр Зозулі має швидший пошук (2 доступи), але гіршу кеш-локальність та складнішу логіку вставки. Фільтр-частка має кращу кеш-локальність, але може вимагати більше порівнянь при довгому скануванні пробіжки.

d) Фільтр-частка не підтримує видалення.

6. Якщо вставка у фільтр-частку призводить до зсуву елементів, чи потрібно оновлювати їхні біти **`is_shifted`**?

a) Ні, цей біт встановлюється один раз і ніколи не змінюється.

b) Так, для всіх зсунутих елементів біт **`is_shifted`** має бути встановлений в 1, оскільки вони тепер не у своїх домашніх комірках.

c) Потрібно оновлювати лише для першого зсунутого елемента.

d) Потрібно оновлювати лише для елементів, чия частка відрізняється.

7. Яке твердження про фільтр-частку є хибним?

a) Він може бути динамічно розширений.

b) Він підтримує видалення елементів.

c) Він гарантує пошук за $O(1)$ у найгіршому випадку.

d) Його продуктивність краща при низьких коефіцієнтах заповнення.

Відповіді.

1. *c)* Метадані є ключовими. **`is_occupied`** позначає початок кластера, а **`is_continuation`** — чи є поточний елемент продовженням пробіжки попереднього. Комбінація цих бітів дозволяє точно визначити межі кожної пробіжки.

2. *b)* Це головна оптимізація. Завдяки метаданим, фільтр-частка точно знає, де закінчується група елементів з однаковою часткою, і не буде сканувати решту кластера, як це робить звичайна хеш-таблиця.

3. *b)* LSM-дерева постійно зливають менші файли в більші. Можливість ефективно об'єднати фільтри, що відповідають цим файлам, є значною перевагою, яку фільтр-частка надає завдяки своїй структурі.

4. *b)* Збільшення **`r`** означає зменшення **`q`**. Менше **`q`** бітів — менше комірок ($m = 2^q$), що збільшує навантаження і кількість колізій. Однак довший залишок **`r`** означає, що менше різних ключів будуть мати однаковий відбиток, що знижує ймовірність помилки при перевірці.

5. *c)* Це ключовий архітектурний компроміс. Фільтр Зозулі робить два непередбачувані стрибки по пам'яті. Фільтр-частка робить один стрибок, а потім послідовно читає пам'ять, що значно краще для кешу процесора.

6. *b)* Мета біта **`is_shifted`** — точно вказувати, чи знаходиться залишок у своїй домашній комірці **`q`**. Будь-який зсув порушує цю умову, тому біт має бути встановлений.

7. *c)* Як і будь-яка структура з лінійним пробуванням, у найгіршому випадку (при високому заповненні та невдалих хешах) пошук може виродитись у сканування значної частини таблиці, що дає складність $O(n)$. Гарантію $O(1)$ дає, наприклад, Cuckoo Hashing.

Завдання для самостійної роботи

1. Ручне трасування. Створіть фільтр-частку з q=3, r=4. Покажіть покроково, як будуть вставлені елементи з хешами (частка | залишок): **`001|0101`**, **`001|1011`**, **`010|0011`**, **`001|0001`**.

2. Реалізація видалення. Додайте до вашої спрощеної реалізації з лабораторної роботи метод `delete`. Врахуйте необхідність зсуву елементів вліво.
3. Порівняння з іншими фільтрами. Напишіть есе, що детально порівнює фільтр-частку з фільтром Блума та фільтром Зозулі за критеріями: використання пам'яті, швидкість вставки, швидкість пошуку, підтримка видалення, складність реалізації.
4. Аналіз метаданих. Детально опишіть, як комбінація трьох бітів метаданих (`is_occupied`, `is_continuation`, `is_shifted`) дозволяє точно відновити межі кожної пробіжки.
5. Ефект заповнення. Проведіть експеримент: виміряйте середню кількість кроків, необхідних для вставки, при коефіцієнті заповнення 50%, 80%, 90% та 95%.
6. Об'єднання фільтрів. Дослідіть та опишіть алгоритм для ефективного об'єднання (merge) двох фільтрів-часток.
7. Динамічне розширення. Опишіть алгоритм, як можна розширити фільтр-частку (наприклад, подвоїти розмір), коли він стає занадто заповненим.
8. Оцінка кардинальності. Чи можна використовувати фільтр-частку для приблизної оцінки кількості унікальних елементів у множині? Якщо так, то як?
9. Оптимізація пам'яті. Подумайте, як можна було б зменшити розмір метаданих. Чи можна обійтися двома бітами замість трьох? Які будуть наслідки?
10. Есе-аналіз. Напишіть есе на тему "Чому кеш-ефективність є"тихим вбивцею" продуктивності і як фільтр-частка вирішує цю проблему краще за інші структури?".
11. Робота з дублікатами. Як поводитиметься фільтр-частка, якщо до нього декілька разів додати один і той самий елемент? Чи потрібні зміни в алгоритмі?
12. Ітерація по елементах. Чи можливо реалізувати ітератор, що проходить по всіх елементах, збережених у фільтрі-частці? Якою буде його складність?

13. Порівняння з хеш-таблицею. Порівняйте фільтр-частку та хеш-таблицю з лінійним пробуванням. Які переваги дає зберігання лише залишків та метаданих?

14. Вибір параметрів `q` та `r`. Напишіть програму, яка для заданої кількості елементів `n` та бажаної ймовірності помилки `p` рекомендує оптимальні значення `q` та `r`.

15. Відновлення елементів. Чи можливо, маючи лише вміст фільтра-частки, відновити початкові елементи, що були до нього додані? Поясніть, чому.

Дерева пошуку

Дерева пошуку

Пошук порядку в хаосі даних.

У попередніх розділах ми детально ознайомилися з лінійними структурами даних. Ми встановили, що, незважаючи на їхню простоту та корисність, вони мають фундаментальний недолік: повільний пошук. У невпорядкованій колекції, щоб знайти потрібний елемент, у найгіршому випадку доведеться переглянути всі записи. Це лінійний пошук зі складністю $O(n)$, і ця "тиранія лінійного часу" робить такі структури неефективними для роботи з великими обсягами даних.

Навіть якщо ми відсортуємо масив і застосуємо швидкий двійковий пошук зі складністю $O(\log_2 n)$, ми заплатимо за це високою вартістю модифікації даних ($O(n)$ для вставки чи видалення).

Перед нами постає ключове питання: чи існують способи організації даних, які дозволяють виконувати і пошук, і вставку, і видалення значно швидше за лінійний час? Чи можемо ми одночасно мати і швидкість, і гнучкість?

Відповідь — так. Історія комп'ютерних наук запропонувала два принципово різні, але однаково потужні підходи до розв'язання цієї проблеми. Перший шлях, який ми розглянемо в наступному великому розділі — це хешування, яке намагається досягти миттєвого доступу шляхом прямого обчислення адреси даних.

Другий шлях, якому присвячений цей розділ — це дерева пошуку. Цей підхід, на відміну від хешування, ставить на перше місце збереження порядку, організовуючи дані не в лінію, а в ієрархічну структуру, що нагадує дерево рішень.

Уявіть гру “вгадай число”:

- Я загадав число від 1 до 100.
- Ваш перший хід: “50?”
- Я: “Менше”.
- Ви миттєво відкинули половину варіантів (всі числа від 50 до 100).
- Ваш другий хід: “25?”
- Я: “Більше”.
- Ви знову відкинули половину з решти варіантів.

Бінарне дерево пошуку (Binary Search Tree, BST) є втіленням цієї логіки. Кожен вузол у дереві є точкою прийняття рішення. Якщо шуканий елемент менший за поточний вузол, ми йдемо вліво; якщо більший — вправо. Кожен такий крок відсікає ціле піддерево варіантів.

Якщо дерево є збалансованим (тобто його гілки мають приблизно однакову довжину), то шлях від кореня до будь-якого листка буде дуже коротким. Для дерева з n елементів його висота буде приблизно дорівнювати $\log_2 n$. Це означає, що операції пошуку, вставки та видалення матимуть гарантовану складність $O(\log_2 n)$.

Звісно, тут є свої виклики. Головна проблема — як підтримувати дерево збалансованим, коли до нього постійно додаються та видаляються елементи? Просте бінарне дерево пошуку може “виродитись” у зв’язний список, втративши всі свої переваги. У цьому розділі ми детально розглянемо механізми самобалансування на прикладі таких структур, як AVL-дерева та червоно-чорні дерева, які автоматично підтримують свою структуру близькою до ідеальної.

У наступних підрозділах ми детально розберемо:

1. Бінарні дерева пошуку (BST): Ми почнемо з простого BST, зрозуміємо його переваги та побачимо його фундаментальний недолік — проблему виродження.

2. AVL-дерева: Ми вивчимо перший класичний алгоритм самобалансування, що базується на строгій підтримці висоти піддерев.

3. Червоно-чорні дерева: Розглянемо інший, більш гнучкий підхід до балансування, який є основою багатьох стандартних бібліотечних реалізацій.

4. B-дерева: Ми вийдемо за межі бінарних дерев і познайомимось зі структурами, які є основою практично всіх сучасних баз даних та файлових систем.

5. Інші дерева із цікавими властивостями.

Розуміння деревних структур є абсолютно необхідним для будь-якого розробника, оскільки вони лежать в основі інструментів, якими ми користуємося щодня.

Бінарні дерева пошуку

Бінарне дерево пошуку — це фундаментальна структура, на якій базується розуміння більш складних деревних алгоритмів. Цей розділ побудований так, щоб провести вас від базової ідеї до повного розуміння всіх його операцій та нюансів.

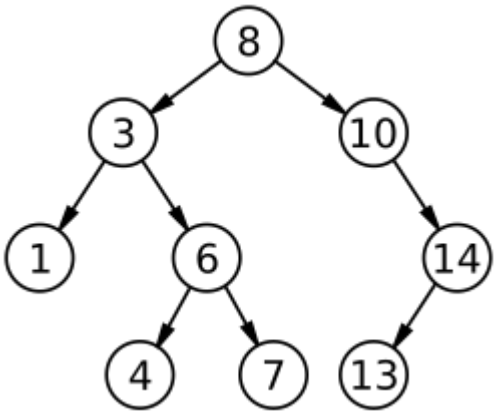
1. Теоретичні відомості

1.1. Ідея та властивості

У вступі до цього розділу ми порівняли пошук впорядкованих даних з грою “вгадай число”. Бінарне дерево пошуку (Binary Search Tree, BST) — це структура даних, яка формалізує цю ідею. Замість того, щоб зберігати дані в лінійній послідовності, як масив, BST організовує їх в ієрархію, де кожен елемент допомагає нам прийняти рішення, куди рухатись далі.

Головна властивість бінарного дерева пошуку:

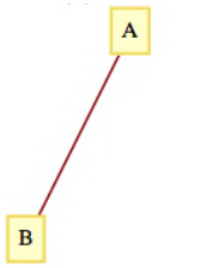
Для абсолютно кожного вузла X у дереві, всі ключі в його лівому піддереві є меншими за ключ вузла X, а всі ключі в його правому піддереві є більшими за ключ вузла X.



Ця проста, але сувора властивість дозволяє відсікати половину простору для пошуку на кожному кроці, що і призводить до високої ефективності.

Але бінарні дерева не завжди такі красиві. На наступних рисунках ви теж побачите бінарні дерева.

Це одне дерево:



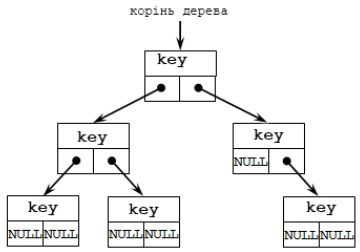
А ца ІНШЕ ДЕРЕВО.

Введемо основні терміни:

- Вузол (Node): Основний будівельний блок, що зберігає ключ та вказівники на нащадків.
- Корінь (Root): Найвищий вузол дерева, з якого починається будь-який обхід.
- Нащадок (Child): Вузол, що знаходиться безпосередньо під іншим вузлом.
- Батько (Parent): Вузол, що знаходиться безпосередньо над іншим вузлом.
- Листок (Leaf): Вузол, що не має нащадків.
- Піддерево (Subtree): Дерево, що складається з нащадка та всіх його нащадків.
- Висота дерева (Height): Максимальна довжина шляху від кореня до найвіддаленішого листка.

1.2. Структура вузла

Будь-яке дерево складається з вузлів. Для бінарного дерева вузол зазвичай містить три частини:

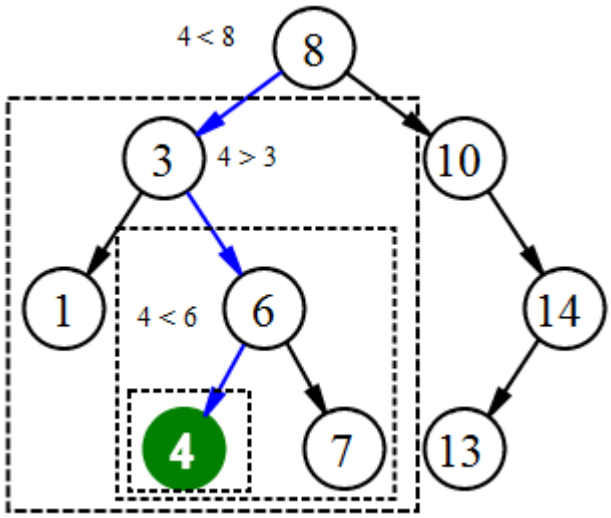



```
// Структура для вузла бінарного дерева
Структура Node:
    key     // Ключ (дані, напр., ціле число, рядок)
    left    // Вказівник на лівого нащадка (інший вузол або NULL)
    right   // Вказівник на правого нащадка (інший вузол або NULL)
```

1.3. Основні операції

Пошук

Пошук у BST — це пряма реалізація ідеї “вгадай число”. Наприклад, в такому дереві ми хочимо знайти число «4». Починаємо проходити по вуздах дерева перевіряючи умову «більше-менше». Якщо «менше», то переходимо до лівого піддерева, якщо «більше – то до «правого». І так повторюємо, поки не знfйдемо вузол із заданим числом, або значення null.




```
// Рекурсивна функція пошуку ключа в дереві з коренем 'node'
ФУНКЦІЯ Search(node, key):
    // Базовий випадок 1: Дійшли до кінця гілки, елемент не знайдено.
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ false
    КІНЕЦЬ ЯКЩО

    // Базовий випадок 2: Елемент знайдено.
    ЯКЩО node.key == key ТО
        ПОВЕРНУТИ true
    КІНЕЦЬ ЯКЩО
```

```
    // Рекурсивний крок: обираємо, в якому піддереві продовжити пошук
.
    ЯКЩО key < node.key ТО
        ПОВЕРНУТИ Search(node.left, key)
    ІНАКШЕ
        ПОВЕРНУТИ Search(node.right, key)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Вставка

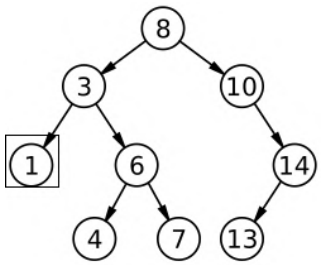
Щоб додати новий елемент, ми спершу знаходимо для нього правильне місце, тоюто шукаємо вільне місце за правилом слідуючи правилам пошуку «більше-менше». Вільне місце – це місце із значенням null у відповідній частині дерева. Коли ми натрапляємо на вільне місце (NULL), ми додаємо туди новий вузол.

Наведемо псевдокод цього алгоритму:

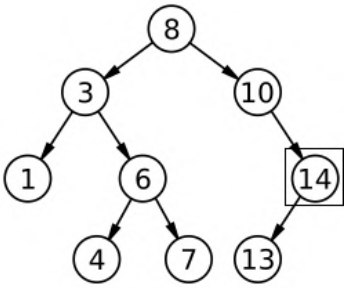
```
// Рекурсивна функція вставки ключа в дерево з коренем 'node'
ФУНКЦІЯ Insert(node, key):
    // Якщо піддерево порожнє, ми знайшли місце для вставки.
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ Створити_Вузол(key)
    КІНЕЦЬ ЯКЩО

    // Рекурсивно спускаємось по дереву
    ЯКЩО key < node.key ТО
        node.left = Insert(node.left, key)
    ІНАКШЕ ЯКЩО key > node.key ТО
        node.right = Insert(node.right, key)
    КІНЕЦЬ ЯКЩО

    // Повертаємо вказівник на вузол (можливо, незмінений)
    ПОВЕРНУТИ node
КІНЕЦЬ ФУНКЦІЇ
```

Пошук мінімуму та максимуму

Завдяки властивостям BST, знайти екстремальні значення дуже легко. Мінімальний елемент — це завжди “найлівіший” вузол,



а максимальний — “найправіший”.



Наведемо псевдокод цього алгоритму:

```
ФУНКЦІЯ findMin(node):
    current = node
    ПОКИ current.left != NULL:
        current = current.left
    КІНЕЦЬ ПОКИ
    ПОВЕРНУТИ current
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ findMax(node):
    current = node
    ПОКИ current.right != NULL:
        current = current.right
    КІНЕЦЬ ПОКИ
    ПОВЕРНУТИ current
КІНЕЦЬ ФУНКЦІЇ
```

Процедура видалення вузла з бінарного дерева пошуку

Видалення вузла — це найскладніша з базових операцій над BST, оскільки ми повинні не просто видалити елемент, а й зберегти фундаментальну властивість дерева: **`ліве_піддерево < корінь < праве_піддерево`**.

Процес завжди починається однаково: знайти вузол, який потрібно видалити. Для цього використовується стандартний алгоритм пошуку. Після того, як вузол знайдено, виникає три можливих сценарії, що залежать від кількості його нащадків.

Випадок 1: Вузол є листком (не має нащадків)

Це найпростіший сценарій.

Аналогія: Уявіть, що ви обрізаєте один листок з гілки дерева. Це не впливає на структуру решти гілок.

Процедура така:

1. Знаходимо батьківський вузол того елемента, який ми видаляємо.

2. Перевіряємо, чи є наш вузол лівим чи правим нащадком для цього батька.

3. Встановлюємо відповідний вказівник (`left` або `right`) батьківського вузла на `NULL`.

4. Звільняємо пам'ять, яку займав вузол, що видаляється.

5. Якщо вузол, що видаляється, є коренем і не має нащадків, то після його видалення дерево просто стає порожнім.

Випадок 2: Вузол має рівно одного нащадка

Цей випадок також є відносно простим.

Аналогія: Уявіть, що ви видаляєте ланку з ланцюжка, яка з'єднана лише з однією наступною ланкою. Ви просто "обрізаєте" її, з'єднуючи попередню ланку напряму з наступною.

Процедура така:

1. Знаходимо батьківський вузол елемента, який ми видаляємо.

2. "Обходимо" вузол, що видаляється, "підвішуючи" його єдиного нащадка на його місце.

    - Якщо наш вузол був лівим нащадком свого батька, то тепер вказівник `left` батька буде вказувати на єдиного нащадка нашого вузла.
    - Якщо наш вузол був правим нащадком, то вказівник `right` батька буде вказувати на цього нащадка.

3. Звільняємо пам'ять, яку займав вузол, що видаляється.

4. Якщо вузол, що видаляється, є коренем, то його єдиний нащадок стає новим коренем дерева.

Випадок 3: Вузол має двох нащадків

Це найскладніший і найцікавіший сценарій. Ми не можемо просто видалити цей вузол, оскільки це розділить дерево на дві незв'язані частини — ліве та праве піддерева.

Аналогія: Це схоже на серйозну хірургічну операцію. Не можна просто “вирізати” орган, потрібно знайти йому заміну, яка зможе виконувати його функції, не порушивши роботу всього організму.

Процедура (з використанням “наступника”): Нам потрібно знайти “спадкоємця” — вузол, який може зайняти місце видаленого, не порушивши властивості BST. Ідеальним кандидатом є наступний за порядком елемент (in-order successor). Це найменший елемент у правому піддереві вузла, що видаляється. Чому саме він? Бо він гарантовано більший за всі елементи в лівому піддереві і менший за всі інші елементи в правому піддереві.

1. Знаходимо вузол **P**, який потрібно видалити.
2. Знаходимо його наступника **S**. Для цього робимо один крок вправо (до **P.right**), а потім рухаємось вліво до кінця (**поки є лівий нащадок**).
3. Копіюємо дані. Ми копіюємо значення (ключ) з вузла-наступника **S** у наш вузол **P**. Тепер вузол **P** має правильне значення, але в дереві з’явився дублікат (оскільки вузол **S** все ще існує).
4. Видаляємо дублікат. Ми рекурсивно викликаємо функцію видалення для ключа, який ми щойно скопіювали, але тепер пошук починається з правого піддерева (**P.right**).

Як це працює? Вузол-наступник **S** (найменший у правому піддереві) за визначенням не може мати лівого нащадка. Отже, його видалення — це завжди простий випадок 1 або 2, який ми вже вміємо обробляти. Таким чином, ми звели складну задачу до простішої.

Розглянемо псевдокод для процедури видалення:

```
ФУНКЦІЯ Delete(node, key):
    ЯКЩО node == NULL ТО ПОВЕРНУТИ node КІНЕЦЬ ЯКЩО

    // Шукаємо вузол для видалення
    ЯКЩО key < node.key ТО
        node.left = Delete(node.left, key)
    ІНАКШЕ ЯКЩО key > node.key ТО
        node.right = Delete(node.right, key)
    // Вузол з ключем 'key' знайдено
    ІНАКШЕ:
        // Випадок 1: вузол не має нащадків або має одного нащадка
        ЯКЩО node.left == NULL ТО
            temp = node.right
            ВИДАЛИТИ node
```

```
            ПОВЕРНУТИ temp
        ІНАКШЕ ЯКЩО node.right == NULL ТО
            temp = node.left
            ВИДАЛИТИ node
            ПОВЕРНУТИ temp
        КІНЕЦЬ ЯКЩО

        // Випадок 2: вузол має двох нащадків
        // Знаходимо наступний за порядком елемент (найменший у право
му піддереві)
        temp_node = findMin(node.right)
        // Копіюємо його ключ у наш вузол
        node.key = temp_node.key
        // Рекурсивно видаляємо цей наступний елемент з правого підде
рева
        node.right = Delete(node.right, temp_node.key)
    КІНЕЦЬ ІНАКШЕ

    ПОВЕРНУТИ node
КІНЕЦЬ ФУНКЦІЇ
```

Обходи дерева

Оскільки дерево є нелінійною структурою даних, не існує єдиного “очевидного” способу відвідати всі його вузли, як у випадку з масивом чи списком, де ми просто рухаємось від початку до кінця. Натомість, існує декілька стандартизованих “маршрутів” або стратегій обходу.

Ці стратегії можна розділити на дві великі родини:

1. Пошук в глибину (Depth-First Search, DFS): Алгоритм намагається заглибитись якомога далі по одній гілці, перш ніж повернутися назад і спробувати іншу.
2. Пошук в ширину (Breadth-First Search, BFS): Алгоритм відвідує всі вузли на поточному рівні (або “поверсі”), перш ніж спускатися на наступний.

Вибір правильного алгоритму обходу є ключовим, оскільки він напряму залежить від задачі, яку ви намагаєтесь розв’язати.

Пошук в глибину (DFS) та його варіації

1. Теоретичні відомості

1.1. Ідея пошуку в глибину: до кінця шляху

Пошук в глибину (Depth-First Search, DFS) — це рух по лабіринту. Ви обираєте один коридор і йдете ним до самого кінця, доки не впретеся в глухий кут. Лише після цього ви повертаєтесь назад до останнього роздоріжжя, де був інший вибір, і починаєте досліджувати новий шлях.

Основна ідея: DFS досліджує одну гілку графа настільки глибоко, наскільки це можливо, перш ніж повернутися назад (зробити “backtrack”) і почати досліджувати іншу гілку.

Ця стратегія є природно рекурсивною. Кожен виклик функції досліджує одного з сусідів, занурюючись все глибше і глибше. Для реалізації цього процесу неявно використовується стек — системний стек викликів функцій. Кожен рекурсивний виклик додає новий фрейм у стек, а повернення з рекурсії — видаляє його. Це забезпечує поведінку LIFO (Last-In, First-Out), яка є протилежною до FIFO-поведінки черги в BFS.

1.2. Загальний алгоритм та його кроки

Розглянемо формальні кроки для обходу графа, представленого списком суміжності.

1. Ініціалізація. Створити структуру для відстеження відвіданих вершин `visited` (напр., масив булевих значень), щоб не зациклюватися.
2. Запуск. Вибрати стартовий вузол `start_node` і викликати для нього рекурсивну функцію-помічник `DFS_Util`.
3. Рекурсивна функція `DFS_Util(u)`:
    a. Позначити поточну вершину `u` як відвідану.
    b. Обробити вершину `u` (напр., вивести її значення).
    c. Для кожного сусіда `v` вершини `u`:
    d. Якщо `v` ще не була відвідана, рекурсивно викликати для неї `DFS_Util(v)`.

Псевдокод для обходу графа:

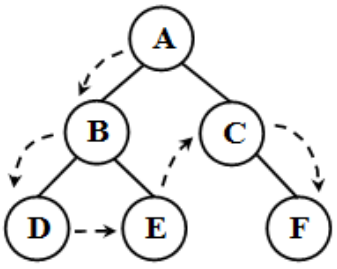
```
// Вхід: graph - граф, N - кількість вершин
// visited - глобальний або переданий за посиланням масив для відстеження
ФУНКЦІЯ DFS_Graph(graph, N):
    visited = новий_масив_булевих(N, заповнений false)
    // Запускаємо обхід для кожної вершини, щоб обійти незв'язні компоненти
    ДЛЯ і ВІД 0 ДО N-1:
        ЯКЩО visited[i] == false ТО
```

```
            DFS_Util(graph, i, visited)
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ФУНКЦІЇ

// Рекурсивна функція-помічник
ФУНКЦІЯ DFS_Util(graph, u, visited):
    // 1. Позначаємо поточну вершину як відвідану
    visited[u] = true
    // 2. Обробляємо її
    Обробити(u)

    // 3. Рекурсивно відвідуємо всіх невідвіданих сусідів
    ДЛЯ кожного v в graph.adjacency_list[u]:
        ЯКЩО visited[v] == false ТО
            DFS_Util(graph, v, visited)
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ФУНКЦІЇ
```

1.3. Варіації для дерев: порядок має значення

Для дерев, де є чіткий порядок нащадків (лівий та правий), DFS розкривається у трьох класичних варіаціях. Вони відрізняються лише моментом, коли ми “обробляємо” сам вузол відносно його піддерев.

1. Прямий обхід (Pre-order Traversal): Корінь → Ліве → Праве Спочатку обробляємо батьківський вузол, потім повністю обробляємо ліве піддерево, а потім — праве.



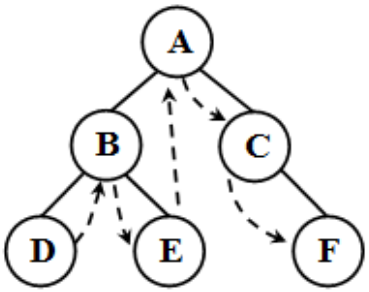
A B D E C F

– Застосування: Створення копії дерева, отримання префіксного запису виразу.

```
ФУНКЦІЯ PreOrder(node):
    ЯКЩО node == NULL ТО ПОВЕРНУТИ КІНЕЦЬ ЯКЩО
    Обробити(node.key)
    PreOrder(node.left)
    PreOrder(node.right)
```

```
КІНЕЦЬ ФУНКЦІЇ
```

2. Симетричний обхід (In-order Traversal): Ліве → Корінь → Праве Спочатку повністю обробляємо ліве піддерево, потім — батьківський вузол, і наостанок — праве піддерево.

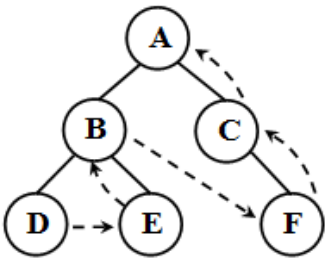


D B E A C F

– Застосування: Для бінарних дерев пошуку (BST) цей обхід відвідує вузли у відсортованому порядку.

```
ФУНКЦІЯ InOrder(node):
    ЯКЩО node == NULL ТО ПОВЕРНУТИ КІНЕЦЬ ЯКЩО
    InOrder(node.left)
    Обробити(node.key)
    InOrder(node.right)
КІНЕЦЬ ФУНКЦІЇ
```

3. Зворотний обхід (Post-order Traversal): Ліве → Праве → Корінь Спочатку повністю обробляємо ліве та праве піддерева, і лише після цього — сам батьківський вузол.



D E B F C A

– Застосування: Видалення дерева з пам’яті (не можна видалити батька, поки не видалені нащадки), отримання постфіксного запису виразу.

```
ФУНКЦІЯ PostOrder(node):
    ЯКЩО node == NULL ТО ПОВЕРНУТИ КІНЕЦЬ ЯКЩО
    PostOrder(node.left)
    PostOrder(node.right)
    Обробити(node.key)
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Аналіз складності

– Часова складність: Як і в BFS, кожен вузол та кожне ребро відвідуються рівно один раз. Загальна складність: $O(V + E)$ (де V — кількість вершин, E — кількість ребер).
– Просторова складність: Тут криється ключова відмінність від BFS. Пам'ять використовується системним стеком для зберігання рекурсивних викликів. У найгіршому випадку (наприклад, для графа, що виродився у довгий ланцюжок), глибина рекурсії може дорівнювати кількості всіх вершин V. Просторова складність: $O(H)$, де H — максимальна глибина рекурсії (або висота дерева пошуку). У найгіршому випадку H може дорівнювати V, тому складність $O(V)$.

2. Область застосування

DFS є потужним інструментом для розв'язання задач, пов'язаних зі структурою графа та шляхами в ньому.

– Пошук шляхів та циклів: DFS чудово підходить для того, щоб знайти будь-який шлях між двома вершинами або визначити, чи містить граф цикл.
– Топологічне сортування: Для орієнтованих ациклічних графів (DAG) DFS дозволяє впорядкувати вершини так, щоб для кожного ребра його початкова вершина йшла раніше за кінцеву.
– Пошук компонент зв'язності: Дозволяє знайти всі зв'язні компоненти в неорієнтованому графі або компоненти сильної зв'язності в орієнтованому.
– Розв'язання головоломок: Пошук виходу з лабіринту є класичним прикладом DFS.

3. Приклади застосування

– Пошук файлів у файловій системі: Утиліти пошуку часто використовують DFS для рекурсивного обходу каталогів.
– Аналіз залежностей: У системах збірки проєктів (як `make` або `Maven`) топологічне сортування на основі DFS використовується для визначення правильного порядку компіляції файлів.
– Штучний інтелект в іграх: Для дослідження дерева можливих ходів у іграх.

Переваги та недоліки

Переваги: Головна перевага DFS — це його ефективність по пам’яті для “широких” графів, де кількість вершин на одному рівні може бути дуже великою. На відміну від BFS, якому доведеться зберігати їх усі в черзі, DFS зберігає в стеку лише один поточний шлях. Його рекурсивна реалізація часто є дуже елегантною та компактною.

Недоліки: Основний недолік полягає в тому, що DFS не гарантує знаходження найкоротшого шляху. Він може заглибитись у дуже довгу гілку, в той час як короткий шлях може знаходитись зовсім поруч зі стартовою вершиною. Також для графів з дуже великою глибиною рекурсії існує ризик переповнення стека викликів (stack overflow).

Контрольні запитання

2. У чому полягає основна відмінність у стратегії дослідження графа між DFS та BFS?
3. Яка структура даних неявно використовується при рекурсивній реалізації DFS?
4. Для якої задачі з трьох варіацій обходу бінарного дерева (прямий, симетричний, зворотний) кожна є найкращим вибором?
5. Чому просторова складність DFS у найгіршому випадку дорівнює $O(V)$? Наведіть приклад графа, для якого це станеться.
6. Поясніть, чому DFS не знаходить найкоротший шлях у незваженому графі.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація обходу графа в глибину (DFS) та пошук циклів.

2. Мета

Практичне засвоєння рекурсивної логіки алгоритму DFS. Модифікація базового алгоритму для розв’язання практичної задачі — виявлення циклу в орієнтованому графі.

3. Завдання

Створити програму, яка для заданого орієнтованого графа визначає, чи містить він цикл.

4. Опис

Для виявлення циклу нам знадобиться відстежувати не тільки відвідані вершини (`visited`), але й вершини, що знаходяться у поточному стеку рекурсії (`recursionStack`). Якщо під час обходу ми зустрічаємо вершину, яка вже є в `recursionStack`, це означає, що ми знайшли зворотнє ребро, а отже, і цикл.

5. Порядок виконання

1. Створіть представлення графа: Реалізуйте клас або структуру для представлення орієнтованого графа у вигляді списку суміжності.
2. Ініціалізуйте структури: Створіть два булеві масиви: `visited` та `recursionStack`.
3. Модифікуйте `DFS_Util`:
    - На вході у функцію для вершини `u` встановлюйте `visited[u] = true` та `recursionStack[u] = true`.
    - У циклі по сусідах `v` перед рекурсивним викликом додайте перевірку: `ЯКЩО recursionStack[v] == true ТО ПОВЕРНУТИ true` (цикл знайдено).
    - На виході з функції (після завершення циклу по сусідах) встановлюйте `recursionStack[u] = false`. Це ключовий крок “повернення назад”.
4. Створіть головну функцію: Напишіть цикл, що запускає ваш модифікований DFS для всіх вершин графа.

6. Результат

Додаток, що для заданого графа коректно визначає наявність чи відсутність циклу.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією графа та алгоритму пошуку циклу.
2. Продемонструвати роботу додатку на двох графах: одному з циклом, іншому — без.
3. Бути готовим відповісти на питання:
    - Пояснити різницю між масивами `visited` та `recursionStack` у вашому коді. Чому одного масиву `visited` недостатньо?

- Чому важливо встановлювати `recursionStack[u] = false` на виході з рекурсії?
- Яка часова складність вашого алгоритму?

Тести

1. Яка структура даних неявно використовується при рекурсивній реалізації DFS?

a) Черга

b) Стек

c) Хеш-таблиця

d) Двозв'язний список

2. Який обхід бінарного дерева потрібно використати для його повного видалення з пам'яті?

a) Прямий (Pre-order)

b) Симетричний (In-order)

c) Зворотний (Post-order)

d) В ширину (Level-order)

3. Яка часова складність DFS для графа з V вершинами та E ребрами?

a) $O(V \cdot E)$

b) $O(V + E)$

c) $O(V \log E)$

d) $O(V^2)$

4. Для якої задачі DFS є менш придатним, ніж BFS?

a) Пошук компонент зв'язності.

b) Пошук циклу в графі.

c) Пошук найкоротшого шляху в незваженому графі.

d) Топологічне сортування.

5. Продовженням якого шляху буде DFS після відвідування глухого кута?

a) Почне з випадкової вершини.

b) Завершить роботу.

c) Повернеться до попередньої вершини, де був нерозглянутий шлях.

d) Повернеться до стартової вершини.

Відповіді на тести: 1-b, 2-c, 3-b, 4-c, 5-c

Тести підвищеної складності

1. Якщо реалізувати DFS ітеративно, яку структуру даних потрібно використовувати явно?

a) Чергу з пріоритетами.

b) Чергу (Queue).

c) Стек (Stack).

d) Хеш-таблицю.

2. У вас є спрямований ациклічний граф (DAG). В якому порядку потрібно вивести вершини після завершення DFS, щоб отримана послідовність була топологічно відсортованою?

a) У порядку їхнього першого відвідування (Pre-order).

b) У зворотному порядку до їхнього першого відвідування.

c) У порядку завершення їхньої обробки (коли всі їхні нащадки вже відвідані).

d) У зворотному порядку до завершення їхньої обробки.

3. При пошуку циклів в неорієнтованому графі за допомогою DFS, як відрізнити ребро, що веде назад до батька, від ребра, що утворює справжній цикл?

a) Ніяк, будь-яке ребро до вже відвіданої вершини утворює цикл.

b) Потрібно передавати в рекурсивний виклик не тільки поточну вершину `u`, але і її батька `parent`, і ігнорувати ребро, що веде назад до `parent`.

c) Потрібно використовувати BFS замість DFS.

d) Потрібно підраховувати кількість ребер.

4. Яка максимальна глибина стека викликів для DFS на повному графі з N вершинами?

a) N

b) 1

c) N/2

d) log*N*

5. “Лісовий” обхід DFS (коли головна функція запускає DFS в циклі для всіх невідвіданих вершин) використовується для:

a) Прискорення роботи на деревах.

b) Роботи з графами, що мають більше одного ребра.

c) Гарантованого обходу всіх вершин у незв’язному графі.

d) Пошуку найкоротшого шляху.

6. Яка варіація обходу дерева є найбільш доцільною для обчислення значення арифметичного виразу, представленого у вигляді дерева виразів?

a) Прямий обхід.

b) Симетричний обхід.

c) Зворотний обхід.

d) Обхід в ширину.

7. У чому полягає основний ризик використання наївної рекурсивної реалізації DFS на практиці?

a) Низька швидкість роботи.

b) Велике споживання оперативної пам’яті для черги.

c) Ризик переповнення системного стека викликів (Stack Overflow) для графів з великою глибиною.

d) Неможливість обробляти графи з циклами.

e) 1. *c)* Ітеративний DFS повністю імітує роботу системного стека викликів. Ми кладемо вершину в стек, потім виймаємо, обробляємо і кладемо в стек всіх її сусідів. Принцип LIFO стека забезпечує заглиблення по одній гілці.

f) *2. d)* Вершина завершує свою обробку лише після того, як були оброблені всі вершини, досяжні з неї. Отже, вершина, що завершилась раніше, знаходиться “правіше” у сортуванні. Щоб отримати правильний порядок (від залежностей до залежних), потрібно взяти послідовність завершення у зворотному порядку.

g) 3. *b)* У неорієнтованому графі ребро `(u, v)` означає, що можна пройти з `u` в `v` і назад. Коли ми з `v` досліджуємо сусідів, ми обов’язково натрапимо на `u`, з якого прийшли. Це не цикл. Щоб відрізнити цей тривіальний випадок, ми просто ігноруємо ребро, що веде назад до безпосереднього предка.

h) 4. *a)* У повному графі кожна вершина з’єднана з кожною. DFS може побудувати шлях, що відвідує кожну вершину рівно один раз, перш ніж повертатися назад. Такий шлях має довжину N, і глибина рекурсії, а отже, і стека викликів, досягне N.

i) *5. c)* Один запуск DFS з довільної вершини обійде лише ту компоненту зв’язності, до якої належить ця вершина. Щоб обійти весь граф, якщо він складається з кількох ізольованих “островів”, потрібно після завершення одного обходу перевірити, чи є ще невідвідані вершини, і запустити для них новий обхід.

j) 6. *c)* Щоб обчислити операцію у вузлі (напр., **+**), потрібно спершу обчислити значення її операндів, які знаходяться в лівому та правому піддеревах. Зворотний обхід (Ліве → Праве → Корінь) гарантує саме такий порядок.

k) 7. *c)* Кожен рекурсивний виклик займає місце в системному стеку. Якщо граф має дуже довгий шлях, глибина рекурсії може перевищити ліміт стека, що призведе до аварійного завершення програми. Для таких випадків використовують ітеративну реалізацію з явним стеком.

Завдання для самостійної роботи

1. Ітеративний DFS. Реалізуйте обхід графа в глибину за допомогою ітеративного підходу з використанням явного стека.
2. Компоненти зв’язності. Напишіть програму, яка використовує DFS для підрахунку кількості компонент зв’язності у неорієнтованому графі.
3. Топологічне сортування. Реалізуйте алгоритм топологічного сортування для спрямованого ациклічного графа на основі DFS.
4. Пошук мостів. Дослідіть та реалізуйте алгоритм пошуку “мостів” (ребер, видалення яких збільшує кількість компонент зв’язності) у графі.
5. Розфарбовування у два кольори. Реалізуйте перевірку графа на дводольність за допомогою DFS.
6. “Відстані” в DFS. Модифікуйте DFS так, щоб він знаходив відстань від стартової вершини до всіх інших. Порівняйте результат з тим, що дає BFS. Чи є цей шлях найкоротшим?
7. Цикли в неорієнтованому графі. Напишіть функцію, що перевіряє наявність циклу в неорієнтованому графі за допомогою DFS, коректно обробляючи ребра до батьківських вершин.

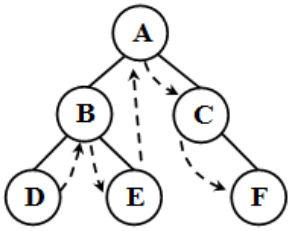
8. Есе-порівняння. Напишіть есе на тему “Рекурсія проти ітерації для реалізації DFS: переваги, недоліки та сфери застосування кожного підходу”.
9. Аналіз дерева обходу. Напишіть програму, яка будує та виводить саме дерево обходу в глибину для заданого графа.
10. Ігрова стратегія. Реалізуйте пошук виграшного шляху у простій грі (напр., хрестики-нулики на полі 3x3) за допомогою DFS, що досліджує дерево ігрових станів.
11. Пошук точок шарніру. Дослідіть та реалізуйте алгоритм пошуку точок шарніру (вершин, видалення яких збільшує кількість компонент зв’язності).
12. Обмежений DFS (Depth-Limited Search). Реалізуйте DFS, який зупиняється, досягнувши певної глибини рекурсії.
13. Ітеративне заглиблення (IDDFS). На основі попереднього завдання, реалізуйте IDDFS — алгоритм, що послідовно запускає DLS зі зростаючою глибиною.
14. Порівняння пам’яті. Проведіть експеримент: згенеруйте два типи графів (довгий “ланцюжок” та широка “зірка”) однакового розміру. Порівняйте максимальний розмір стека/черги, що використовується DFS та BFS для їх обходу.
15. Сильно зв’язні компоненти. Дослідіть та реалізуйте алгоритм Тар’яна або Косараджу для пошуку сильно зв’язних компонент в орієнтованому графі.

Пошук в глибину (DFS) та його варіації

DFS найпростіше реалізувати за допомогою рекурсії. Існує три основні варіації, які відрізняються лише порядком, в якому вони “відвідують” (обробляють) кореневий вузол відносно його лівого та правого піддерев.

1. Симетричний обхід (In-order Traversal)

Порядок: Ліве піддерево → Корінь → Праве піддерево.



D B E A C F

Ідея: Алгоритм спершу повністю обробляє все ліве піддерево, потім відвідує поточний вузол, і лише після цього переходить до обробки всього правого піддерева.

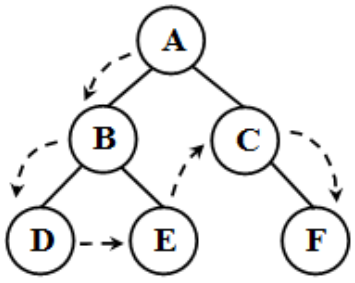
Ключове застосування: Для бінарних дерев пошуку (BST) цей обхід є “магічним”, оскільки він завжди відвідує вузли у відсортованому за зростанням порядку.

Псевдокод:

```
// Вхід: 'node' - вказівник на корінь дерева або піддерева
ФУНКЦІЯ InOrderTraversal(node):
    // Базовий випадок рекурсії: якщо вузол не існує, нічого не робит
и.
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // 1. Рекурсивно обходимо ліве піддерево
    InOrderTraversal(node.left)

    // 2. Відвідуємо (обробляємо) поточний вузол
    Обробити(node.key)

    // 3. Рекурсивно обходимо праве піддерево
    InOrderTraversal(node.right)
КІНЕЦЬ ФУНКЦІЇ
```

2. Прямий обхід (Pre-order Traversal)

Порядок: Корінь → Ліве піддерево → Праве піддерево.



A B D E C F

Ідея: Алгоритм спочатку відвідує поточний вузол, а вже потім рекурсивно обробляє його ліве та праве піддерева.

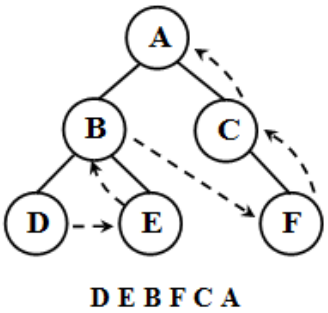
Ключове застосування:

- Створення копії дерева: Цей порядок дозволяє легко відтворити структуру. Ви створюєте копію кореневого вузла, потім рекурсивно копіюєте ліве піддерево і приєднуєте його, а потім — праве.
- Отримання префіксної нотації з дерева виразів.

Псевдокод:

```
// Вхід: 'node' - вказівник на корінь дерева або піддерева
ФУНКЦІЯ PreOrderTraversal(node):
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // 1. Відвідуємо поточний вузол
    Обробити(node.key)

    // 2. Рекурсивно обходимо ліве піддерево
    PreOrderTraversal(node.left)

    // 3. Рекурсивно обходимо праве піддерево
    PreOrderTraversal(node.right)
КІНЕЦЬ ФУНКЦІЇ
```

3. Зворотний обхід (Post-order Traversal)



D E B F C A

Пошук в ширину (BFS) або обхід по рівнях

Тепер настав час познайомитись з його фундаментальною альтернативою — пошуком в ширину. Цей метод досліджує структуру не “занурюючись”, а методично, шар за шаром, що робить його незамінним для цілого класу задач.

1. Теоретичні відомості

1.1. Ідея пошуку в ширину: хвилі на воді

Якщо пошук в глибину (DFS) можна порівняти з рухом по лабіринту, де ми завжди намагаємось пройти якомога далі одним шляхом, то пошук в ширину (Breadth-First Search, BFS) працює за зовсім іншим принципом.

Аналогія: Уявіть, що ви кинули камінь у спокійний ставок. Від нього починають розходитись концентричні кола. Спочатку хвиля досягає всіх найближчих до центру точок, потім вона розширюється і досягає наступного "шару" точок, і так далі.

BFS досліджує граф або дерево аналогічно:

1. Починаємо з початкового вузла (рівень 0).
2. Відвідуємо всіх його безпосередніх сусідів (рівень 1).
3. Лише після цього переходимо до відвідування всіх сусідів вузлів з рівня 1 (тобто, до вузлів на рівні 2).
4. Процес продовжується, доки не будуть відвідані всі досяжні вузли, рівень за рівнем.

Ця стратегія гарантує, що ми знаходимо вузли у порядку їхньої віддаленості від стартової точки. Це робить BFS ідеальним для пошуку найкоротших шляхів у незважених графах.

1.2. Черга (Queue): серце алгоритму BFS

На відміну від DFS, який зручно реалізовувати за допомогою рекурсії (що неявно використовує стек викликів), BFS за своєю природою є ітеративним. Щоб підтримувати правильний порядок відвідування "рівень за рівнем", йому потрібен інструмент, що працює за принципом FIFO (First-In, First-Out). Як ми знаємо з попередніх розділів, цей інструмент — черга (Queue).

Як черга забезпечує роботу BFS:

1. Ми кладемо стартовий вузол у чергу.
2. На кожному кроці ми вилучаємо один вузол з початку черги.
3. Всіх його сусідів ми додаємо в кінець черги.

Це гарантує, що ми не почнемо обробляти сусідів з другого рівня, доки не обробимо всі вузли першого рівня, які були додані до черги раніше.

1.3. Алгоритм та його кроки

Розглянемо формальні кроки для обходу графа, представленого списком суміжності.

1. Ініціалізація. Створити порожню чергу `Q` та структуру для відстеження відвіданих вершин **`visited`** (наприклад, масив булевих значень або хеш-множину), щоб не зациклюватися і не обробляти один вузол двічі.
2. Початковий вузол. Обрати стартовий вузол **`start_node`**, додати його до черги `Q` і позначити як відвіданий у **`visited`**.
3. Головний цикл. Поки черга `Q` не порожня:
    e. Вилучити вузол. Вилучити перший вузол з черги. Назвемо його **`current_node`**.
    f. Обробити вузол. Виконати потрібну дію з **`current_node`** (напр., вивести його значення).
    g. Додати сусідів. Для кожного сусіда **`neighbor`** вузла **`current_node`**:
    h. Якщо **`neighbor`** ще не був відвіданий (перевіряємо у **`visited`**), то:
    ii. Позначити **`neighbor`** як відвіданий.
    iii. Додати **`neighbor`** в кінець черги `Q`.

1.4. Псевдокод

```
// Вхід: graph - граф, представлений списком суміжності,
//       start_node - вершина, з якої починається обхід.
ФУНКЦІЯ BFS(graph, start_node):
    // Кількість вершин у графі
    N = кількість_вершин(graph)

    // Структура для відстеження відвіданих вершин
    visited = новий_масив_булевих(N, заповнений false)

    // 1. Створюємо чергу
    queue = нова_Черга()

    // 2. Додаємо стартовий вузол і позначаємо його
    Додати_в_чергу(queue, start_node)
    visited[start_node] = true

    // 3. Головний цикл
    ПОКИ черга_не_порожня(queue):
        // a. Вилучаємо вузол з початку черги
        current_node = Вилучити_з_черги(queue)
```

```
        // b. Обробляємо його
        Обробити(current_node)

        // c. Додаємо всіх невідвіданих сусідів
        ДЛЯ кожного neighbor в graph.adjacency_list[current_node]:
            ЯКЩО visited[neighbor] == false ТО
                // Позначаємо сусіда як відвіданого
                visited[neighbor] = true
                // Додаємо його в кінець черги
                Додати_в_чергу(queue, neighbor)
            КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ПОКИ
КІНЕЦЬ ФУНКЦІЇ
```

1.5. Аналіз складності

- Часова складність: Кожен вузол додається до черги та вилучається з неї рівно один раз. Це дає нам V операцій (де V — кількість вершин). Коли ми обробляємо вузол, ми переглядаємо всі його ребра. Оскільки кожне ребро в графі має дві вершини, ми переглянемо кожне ребро не більше двох разів (по одному разу з кожного боку). Це дає нам $2E^*$ операцій (де E — кількість ребер). Загальна складність є сумою цих дій: $O(V + E)$. Для дерев, де $E = V - 1$, складність можна спростити до $O(V)$.
- Просторова складність: У найгіршому випадку черга може містити всі вузли одного рівня. Для “широкого” графа (наприклад, повного бінарного дерева) останній рівень може містити близько $V/2$ вузлів. Тому просторова складність оцінюється як $O(V)$ (або, точніше, $O(W)$, де W — максимальна ширина графа).

2. Область застосування

BFS є фундаментальним алгоритмом з широким спектром застосувань.

- Пошук найкоротших шляхів: У незважених графах BFS гарантовано знаходить шлях з мінімальною кількістю ребер.
- Аналіз мереж: Використовується для “картографування” мереж, наприклад, для пошуку всіх вузлів, досяжних з певної точки.
- Веб-краулінг: Пошукові роботи обходять Інтернет, переходячи з однієї сторінки на всі сторінки, на які вона посилається, що є класичним BFS.
- Теорія графів: Перевірка графа на дводольність, пошук компонент зв’язності.

– Штучний інтелект: У деяких алгоритмах пошуку розв’язків.

3. Приклади застосування

– Соціальні мережі: Функція “знайти друзів друзів” (друзі 2-го рівня) — це, по суті, виконання BFS на два рівні вглиб від вашого профілю.
– GPS-навігатори: При пошуку маршруту в місті, де кожне перехрестя — це вершина, а кожна вулиця — ребро, BFS може знайти маршрут з найменшою кількістю поворотів.
– Ігри: Пошук найкоротшого шляху для персонажа на карті, представленій у вигляді сітки.

Переваги та недоліки

Переваги: Основна перевага BFS — це те, що він гарантовано знаходить найкоротший шлях (за кількістю ребер) у незваженому графі. На відміну від рекурсивного DFS, він не ризикує отримати переповнення стека викликів на дуже “довгих”, але вузьких графах. Алгоритм є повним (завжди знаходить розв’язок, якщо він існує) та оптимальним для пошуку найкоротшого шляху.

Недоліки: Головний недолік — це вимоги до пам’яті. Для графів, які є дуже “широкими”, черга може розростися до величезних розмірів, зберігаючи велику кількість вузлів одного рівня. У таких випадках DFS, який зберігає лише один шлях у стеку, може бути значно економнішим по пам’яті. Крім того, якщо потрібно просто знайти *будь-який* шлях, а не найкоротший, і цей шлях знаходиться глибоко в одній гілці, DFS може знайти його швидше, ніж BFS, який спершу методично дослідить усі ближчі рівні.

Контрольні запитання

1. У чому полягає основна відмінність у стратегії дослідження графа між BFS та DFS?
2. Яка структура даних є ключовою для реалізації BFS і чому?
3. Поясніть, чому BFS завжди знаходить найкоротший шлях у незваженому графі.
4. Що описує часова складність $O(V + E)$? Поясніть обидва компоненти.
5. Наведіть приклад задачі, де DFS буде кращим вибором, ніж BFS, і навпаки.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація обходу графа в ширину (BFS) та пошук найкоротшого шляху.

2. Мета

Практичне засвоєння логіки алгоритму BFS, включаючи роботу з чергою та масивом відвіданих вершин. Модифікація базового алгоритму для розв'язання задачі пошуку найкоротших відстаней.

3. Завдання

Створити програму, яка для заданого незваженого графа та стартової вершини знаходить найкоротші відстані (у кількості ребер) до всіх інших досяжних вершин.

4. Опис

Граф можна представити у вигляді списку суміжності. Програма має реалізувати функцію BFS, яка під час обходу буде заповнювати масив відстаней.

5. Порядок виконання

1. Створіть представлення графа: Реалізуйте клас або структуру для представлення графа у вигляді списку суміжності.
2. Ініціалізуйте структури: У функції `BFS_ShortestPath` створіть чергу `Q`, масив `visited` та масив `distance`, ініціалізований нескінченністю (або -1).
3. Початковий вузол: Додайте стартовий вузол `start_node` до черги, позначте його як відвіданий та встановіть `distance[start_node] = 0`.
4. Модифікуйте головний цикл: Всередині циклу `while`, коли ви додаєте невідвіданого сусіда `neighbor` до черги, оновлюйте його відстань: `distance[neighbor] = distance[current_node] + 1`.
5. Виведіть результат: Після завершення роботи циклу, виведіть масив `distance`.

6. Результат

Додаток, що для заданого графа та стартової вершини коректно обчислює та виводить найкоротші відстані до всіх інших вершин.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією графа та модифікованого алгоритму BFS.

2. Продемонструвати роботу додатку на кількох тестових графах.
3. Бути готовим відповісти на питання:
    - Пояснити, чому саме в момент додавання сусіда до черги ми встановлюємо його відстань.
    - Яка роль масиву **`visited`** у вашому алгоритмі? Чи можна обійтися без нього, використовуючи лише масив **`distance`**?
    - Як зміниться алгоритм, якщо граф буде незв'язним?

Тести

1. Яка структура даних є центральною для реалізації BFS?

a) Стек

b) Черга

c) Хеш-таблиця

d) Масив

2. BFS гарантовано знаходить найкоротший шлях у:

a) Будь-якому графі.

b) Лише у деревах.

c) Зважених графах.

d) Незважених графах.

3. Яка часова складність BFS для графа з V вершинами та E ребрами?

a) $O(V \cdot E)$

b) $O(V + E)$

c) $O(V \log E)$

d) $O(V^2)$

4. Якщо граф є повним (кожна вершина з'єднана з кожною), яка буде часова складність BFS?

a) $O(V)$

b) $O(\log V)$

c) $O(V^2)$

d) $O(E)$

5. BFS є прикладом:

a) Жадібного алгоритму.

b) Алгоритму “Розділяй і володарюй”.

c) Повного перебору, що досліджує граф рівень за рівнем.

d) Динамічного програмування.

Відповіді на прості тести: 1-b, 2-d, 3-b, 4-c (оскільки у повному графі $E \approx V^2$, і цей член домінує), 5-c

Тести підвищеної складності

1. В якому випадку просторова складність BFS ($O(V)$) буде значно більшою за просторову складність DFS (O(висота))?

a) Для графа, що вироджується в довгий ланцюжок (список).

b) Для широкого, “кущоподібного” графа, де одна вершина має дуже багато сусідів.

c) Просторова складність DFS завжди більша.

d) Їхня просторова складність завжди однакова.

2. Вам потрібно перевірити, чи є неорієнтований граф дводольним (bipartite). Який алгоритм для цього краще підходить?

a) DFS, оскільки він швидше знайде перший-ліпший шлях.

b) BFS, оскільки обхід по рівнях дозволяє легко “розфарбовувати” вершини у два кольори і перевіряти конфлікти.

c) Алгоритм Дейкстри.

d) Цю задачу неможливо розв’язати за допомогою обходів.

3. Яким буде результат роботи BFS на незв’язному графі, якщо запустити його з однієї вершини?

a) Алгоритм видасть помилку.

b) Алгоритм обійде лише ту компоненту зв’язності, в якій знаходиться стартова вершина.

c) Алгоритм обійде всі вершини у всіх компонентах.

d) Алгоритм увійде в нескінченний цикл.

4. Як можна знайти діаметр дерева (найдовший шлях між двома будь-якими вершинами) за допомогою BFS?

a) Запустити BFS з кожної вершини і знайти максимум з усіх відстаней.

b) Запустити BFS з довільної вершини `u`, знайти найвіддаленішу від неї вершину `v`, а потім запустити BFS з вершини `v`. Довжина шляху до найвіддаленішої від `v` вершини і буде діаметром.

c) Це NP-складна задача, що не розв'язується за допомогою BFS.

d) Потрібно використовувати алгоритм Флойда-Воршелла.

5. Якщо реалізувати BFS за допомогою стека замість черги, що ми отримаємо?

a) Той самий обхід в ширину, але в іншому порядку.

b) Обхід в глибину (DFS).

c) Алгоритм перестане працювати коректно і може зациклитися.

d) Алгоритм буде знаходити найдовший шлях замість найкоротшого.

6. У зваженому графі з додатними вагами ребер для пошуку найкоротшого шляху потрібно використовувати алгоритм Дейкстри. Чим він концептуально схожий на BFS?

a) Він також використовує стек.

b) Він також має складність $O(V + E)$.

c) Він також досліджує вершини "пошарово", але шари визначаються не кількістю ребер, а відстанню від старту. Замість звичайної черги використовується черга з пріоритетами.

d) Жодної схожості немає.

7. Ви реалізуєте BFS і забули використовувати масив `visited`. У якому типі графа це гарантовано призведе до нескінченного циклу?

a) У будь-якому графі, що містить цикл.

b) Лише у повному графі.

c) У будь-якому неорієнтованому графі з більш ніж одним ребром.

d) У спрямованому ациклічному графі (DAG).

Відповіді на тести підвищеної складності:

1. b) У довгому ланцюжку черга BFS ніколи не буде містити більше одного елемента, тоді як стек DFS буде містити весь шлях. У широкому графі черга BFS на певному етапі може містити майже всі вершини, тоді як стек DFS буде містити лише один шлях від кореня до листка.

2. b) BFS ідеально підходить для розфарбовування у два кольори. Починаючи з першої вершини (колір 1), всіх її сусідів фарбуємо в колір 2, їхніх сусідів — назад в колір 1, і так далі. Якщо на якомусь етапі ми зустрічаємо сусіда, який вже пофарбований у той самий колір, граф не є дводольним.

3. b) BFS досліджує лише досяжні з початкової вершини вузли. Щоб обійти весь незв'язний граф, потрібно запустити BFS у циклі для кожної ще не відвіданої компоненти.

4. b) Це стандартний двопрохідний алгоритм для знаходження діаметра дерева. Перший BFS з довільної точки знаходить одну з кінцевих точок найдовшого шляху. Другий BFS з цієї точки гарантовано знаходить іншу кінцеву точку, а отже, і сам діаметр.

5. b) Заміна черги (FIFO) на стек (LIFO) фундаментально змінює стратегію обходу. Замість того, щоб досліджувати найстаріший доданий вузол (сусідній), алгоритм буде досліджувати найновіший, тобто заглиблюватиметься по одній гілці. Це і є визначенням DFS.

6. c) Це ключова аналогія. Алгоритм Дейкстри — це узагальнення BFS для зважених графів. Він так само "розширює фронт" відвіданих вершин, але обирає для наступного кроку не просто будь-якого сусіда, а найближчого до старту.

7. a) У будь-якому графі, що містить цикл (навіть `A -> B -> A`), відсутність перевірки `visited` призведе до нескінченної подорожі. Вузол A додасть в чергу B, потім B буде вилучено і додасть в чергу A, і так далі.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте алгоритм BFS для пошуку найкоротших шляхів згідно з методичними вказівками.
2. Відновлення шляху. Модифікуйте вашу програму так, щоб вона не тільки знаходила довжину шляху, але й виводила сам найкоротший шлях (послідовність вершин). (Підказка: для цього потрібно зберігати "батьків" для кожної вершини).
3. Пошук компонент зв'язності. Напишіть програму, яка використовує BFS для підрахунку кількості компонент зв'язності у неорієнтованому графі.
4. Перевірка на дводольність. Реалізуйте алгоритм перевірки графа на дводольність за допомогою BFS та розфарбовування у два кольори.

5. BFS для матриці. Адаптуйте алгоритм BFS для пошуку найкоротшого шляху в матриці (сітці), де є перешкоди.
6. “Хід коня”. Розв’яжіть задачу про знаходження мінімальної кількості ходів, за яку шаховий кінь може потрапити з однієї клітинки дошки на іншу.
7. Дерева. Порівняйте результати обходу в ширину та будь-якого обходу в глибину для бінарного дерева.
8. Діаметр дерева. Реалізуйте двопрохідний алгоритм на основі BFS для знаходження діаметра дерева.
9. Ітеративне заглиблення (IDDFS). Дослідіть алгоритм IDDFS. Як він поєднує переваги DFS (пам’ять) та BFS (найкоротший шлях)?
10. Двонаправлений пошук. Дослідіть ідею двонаправленого пошуку, коли BFS запускається одночасно зі стартової та кінцевої вершин. Які переваги це дає?
11. Есе-порівняння. Напишіть есе на тему “DFS проти BFS: коли і чому обирати певний тип обходу?”.
12. 0-1 BFS. Дослідіть цю варіацію BFS для графів, де вага ребер може бути або 0, або 1. Яка структура даних використовується замість звичайної черги?
13. Пошук циклу. Реалізуйте алгоритм пошуку циклу в неорієнтованому графі за допомогою BFS.
14. Топологічне сортування. Чи можна виконати топологічне сортування за допомогою BFS? Якщо так, опишіть алгоритм (алгоритм Кана).
15. Візуалізатор. Створіть просту графічну візуалізацію, що показує, в якому порядку BFS та DFS відвідують вершини графа.

1.5. Проблема виродження

Найбільша слабкість простого BST полягає в тому, що його ефективність критично залежить від порядку вставки елементів. Якщо вставляти елементи у вже відсортованому порядку (напр., 10, 20, 30, 40), дерево вироджується у звичайний зв’язний список. У цьому випадку висота дерева стає n, а складність всіх операцій падає до $O(n)$. Це нівелює всі переваги дерева. Саме для боротьби з цією проблемою були створені самозбалансовані дерева (AVL, червоно-чорні), які ми розглянемо далі.

2. Область застосування

- Реалізація АТД “Словник” та “Множина”: BST є природною структурою для їх створення (`map`, `set`).
- Символьні таблиці в компіляторах: Для швидкого пошуку та додавання ідентифікаторів.
- Індексація в базах даних: Хоча на практиці використовуються більш складні B-дерева, вони базуються на ідеях BST.

3. Приклади застосування

- Словник для перевірки орфографії: Програма може зберігати мільйони слів у BST і швидко перевіряти наявність введеного слова.
- Система автодоповнення: При введенні тексту система може швидко знаходити у BST всі слова, що починаються з введеного префікса.

Аналіз складності операцій у бінарному дереві пошуку

Ми вже розглянули, як виконуються основні операції в BST. Тепер настав час провести їх глибокий аналіз. Ефективність бінарного дерева пошуку, як ми побачимо, драматично залежить від його форми, що призводить до значної різниці між середнім та найгіршим випадками.

Ключовим параметром, від якого залежить складність, є висота дерева (h). Це довжина найдовшого шляху від кореня до листка.

Пошук

Ідея аналізу: Кожна операція порівняння в алгоритмі пошуку спускає нас на один рівень глибше в дерево. У найгіршому випадку кількість порівнянь дорівнюватиме висоті дерева.

- Найкращий випадок: $O(1)$ Нам неймовірно пощастило, і елемент, який ми шукаємо, є коренем дерева. Ми знаходимо його за одне порівняння.
- Середній випадок: $O(\log n)$ Якщо ми вставляємо елементи у випадковому порядку, дерево, ймовірно, буде відносно “кущистим” та збалансованим. У такому випадку його висота h буде пропорційною логарифму від кількості вузлів n ($h \approx \log_2 n$). Отже, середня складність пошуку є логарифмічною.
- Найгірший випадок: $O(n)$ Це сценарій виродженого дерева. Якщо ми вставляємо елементи у вже відсортованому порядку, дерево перетворюється на довгий ланцюжок, схожий на зв’язний список.

Його висота стає рівною *n-1*. Щоб знайти останній елемент у такому ланцюжку, нам доведеться пройти через всі *n* вузлів.

Вставка

Ідея аналізу: Операція вставки складається з двох етапів:

1. Пошук відповідного місця для нового вузла.

2. Приєднання нового вузла, що вимагає зміни одного вказівника.

Другий етап виконується за константний час, $O(1)$. Отже, загальна складність операції повністю визначається вартістю першого етапу — пошуку.

- Найкращий випадок: $O(\log n)$ (не $O(1)$!) На відміну від пошуку, ми не можемо випадково вставити елемент у корінь. Нам завжди потрібно спуститися до листка, щоб знайти вільне місце. У збалансованому дереві шлях до будь-якого листка має довжину приблизно $\log n$.
- Середній випадок: $O(\log n)$ Для випадково побудованого дерева шлях до місця вставки також буде логарифмічним.
- Найгірший випадок: $O(n)$ У виродженому дереві нам доведеться пройти весь ланцюжок з n вузлів, щоб знайти місце для вставки в самому кінці.

Видалення

Ідея аналізу: Складність видалення, як і вставки, залежить від початкового пошуку вузла та, у найскладнішому випадку, від додаткового пошуку його наступника.

- Випадок 1 та 2 (вузол має 0 або 1 нащадка): Складність визначається пошуком вузла, який потрібно видалити. Після знаходження, сама операція видалення вимагає лише декількох операцій зі зміни вказівників, що виконуються за $O(1)$.
  - Середня складність: $O(\log n)$
  - Найгірша складність: $O(n)$
- Випадок 3 (вузол має 2 нащадків):
  4. Пошук вузла P для видалення: $O(h)$, де h — висота.
  5. Пошук його наступника S: Наступник — це найменший елемент у правому піддереві P. У найгіршому випадку цей пошук також може зайняти час, пропорційний висоті піддерева, тобто $O(h)$.

6. Видалення S та копіювання: $O(1)$. Загальна складність визначається двома операціями пошуку, що дає $O(h) + O(h) = O(h)$.

Отже, для всіх трьох випадків складність операції видалення така:

- Середня складність: $O(\log n)$
- Найгірша складність: $O(n)$

Пошук мінімуму та максимуму

Ідея аналізу: Щоб знайти мінімальний елемент, ми завжди рухаємось вліво від кореня. Щоб знайти максимальний — завжди вправо.

– Середній випадок: $O(\log n)$ У збалансованому дереві довжина найлівішої та найправішої гілок пропорційна $\log n$.

– Найгірший випадок: $O(n)$ У виродженому дереві (наприклад, відсортованому за зростанням), щоб знайти мінімальний елемент, достатньо одного кроку. Але щоб знайти максимальний, доведеться пройти весь ланцюжок з n елементів.

Обходи дерева

Ідея аналізу: Незалежно від типу обходу та форми дерева, ми повинні відвідати кожен вузол рівно один раз.

– Часова складність: Оскільки ми маємо виконати певну константну роботу для кожного з n вузлів, загальна складність будь-якого повного обходу завжди буде $O(n)$.

– Просторова складність:

- Для рекурсивних обходів в глибину (DFS): Просторова складність визначається максимальною глибиною стека рекурсії. У збалансованому дереві це буде $O(\log n)$, а у виродженому — $O(n)$.
- Для обходу в ширину (BFS): Просторова складність визначається максимальним розміром черги. У найгіршому випадку (для повного, “кущистого” дерева) в черзі може опинитися близько $n/2$ листків. Тому просторова складність BFS — $O(n)$.

Гаразд. Ось детальний аналіз складності для всіх операцій в AVL-дереві, що враховує середній та найгірший випадки.

Переваги та недоліки

Переваги: BST є відносно простою в реалізації структурою, яка в середньому забезпечує дуже хорошу продуктивність. Для випадкових даних операції пошуку, вставки та видалення виконуються за логарифмічний час $O(\log n)$. Важливою перевагою над хеш-таблицями є те, що BST зберігає порядок елементів, що дозволяє легко виконувати обхід у відсортованому порядку, знаходити мінімальний/максимальний елементи та робити діапазонні запити.

Недоліки: Головний і критичний недолік — це відсутність механізму самобалансування. Це робить BST вразливим до "поганих" послідовностей вхідних даних (напр., відсортованих), що призводить до виродження дерева і падіння продуктивності всіх операцій до лінійного часу $O(n)$. Саме тому в промислових системах майже завжди використовують його самозбалансовані аналоги.

Контрольні запитання

1. Сформулюйте основну властивість бінарного дерева пошуку.
2. Чому симетричний обхід BST завжди повертає відсортовану послідовність?
3. Опишіть три випадки, які потрібно розглянути при видаленні вузла з BST.
4. Що таке "виродження" бінарного дерева пошуку і яка послідовність вставки елементів до нього призводить?
5. Які переваги BST має над хеш-таблицею?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація бінарного дерева пошуку та його основних операцій.

2. Мета

Практичне засвоєння логіки роботи з деревними структурами даних, реалізація рекурсивних алгоритмів обходу, вставки та видалення.

3. Завдання

Створити програму, що реалізує бінарне дерево пошуку для зберігання цілих чисел та надає користувачеві меню для виконання основних операцій.

4. Опис

Програма має складатися з класу/структури для вузла (`Node`) та основного класу `BinarySearchTree`, що інкапсулює логіку роботи з деревом.

5. Порядок виконання

1. Створіть структуру `Node`.
2. Створіть клас `BinarySearchTree` з полем `root`.
3. Реалізуйте `insert(key)`: Напишіть публічний метод, який викликає приватний рекурсивний метод, керуючись псевдокодом.
4. Реалізуйте `search(key)`: Аналогічно реалізуйте пошук.
5. Реалізуйте `delete(key)`: Це найскладніша частина. Реалізуйте її, використовуючи допоміжну функцію `findMin`.
6. Реалізуйте обходи: Додайте методи для симетричного, прямого та зворотного обходів.
7. Протестуйте: Створіть у `main` меню та перевірте роботу всіх операцій. Особливу увагу приділіть видаленню вузлів з 0, 1 та 2 нащадками. Перевірте випадок виродження дерева.

6. Результат

Повністю функціональний консольний додаток, що коректно виконує всі операції над BST.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією всіх необхідних методів.
2. Продемонструвати роботу додатку, показавши різні сценарії, включаючи видалення вузла з двома нащадками та виродження дерева.
3. Бути готовим відповісти на питання:
    - Пояснити логіку рекурсії у вашому коді.
    - Як працює ваш алгоритм видалення вузла з двома нащадками?
    - Яка часова та просторова складність реалізованих вами операцій?

Тести BST

1. Який обхід BST виведе всі елементи у відсортованому за зростанням порядку?

а) Прямий (Pre-order)

b) Зворотний (Post-order)

c) Симетричний (In-order)

d) В ширину (Level-order)

2. Яка часова складність пошуку елемента в збалансованому BST з N вузлами?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(N \log N)$

3. Якщо вставляти елементи у відсортованому порядку (1, 2, 3, 4, 5) в пусте BST, яка буде висота отриманого дерева?

a) $\log_2 5$

b) 2

c) 4

d) 5

4. Вузол, що не має нащадків, називається:

a) Корінь

b) Листок

c) Батько

d) Гілка

5. При видаленні вузла з двома нащадками, його зазвичай замінюють на:

a) Його лівого нащадка.

b) Найбільший елемент з правого піддерева.

c) Випадковий елемент з будь-якого піддерева.

d) Найменший елемент з правого піддерева.

Відповіді на прості тести: 1-c, 2-b, 3-c (висота N-1, тобто 4 для 5 елементів), 4-b, 5-d.

Тести підвищеної складності BST

1. Яка часова складність знаходження наступного за величиною елемента (in-order successor) для заданого вузла у збалансованому BST з N вузлами?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) O(висота правого піддерева)

2. Видалення якого з елементів з добре збалансованого BST є обчислювально “найдорожчим” у найгіршому випадку?

a) Листка.

b) Вузла, що має лише одного лівого нащадка.

c) Вузла, що має лише одного правого нащадка.

d) Кореневого вузла, що має двох нащадків.

3. Яка з операцій НЕ може бути виконана за час, кращий ніж $O(N)$, у виродженому BST?

a) Вставка нового елемента.

b) Пошук мінімального елемента.

c) Пошук максимального елемента.

d) Всі перелічені.

4. Використовуючи лише зворотний (`Post-order`) та симетричний (`In-order`) обходи, чи можна однозначно відновити структуру бінарного дерева?

a) Так, завжди.

b) Ні, ніколи.

c) Так, але лише якщо дерево є повним.

d) Так, але для цього потрібен прямий (`Pre-order`) та симетричний обходи.

5. Яка просторова складність рекурсивної реалізації обходу в глибину (DFS) для BST з N вузлами у найгіршому випадку?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(N^2)$

6. Який з алгоритмів неявно використовується при видаленні вузла з двома нащадками?

a) Пошук максимального елемента.

b) Пошук в ширину.

c) Алгоритм Дейкстри.

d) Пошук наступного елемента (successor).

7. Побудова BST з N елементів шляхом послідовної вставки випадкових ключів має очікувану часову складність:

a) $O(N^2)$

b) $O(N\log N)$

c) $O(N)$

d) $O(\log N)$

Відповіді на тести підвищеної складності:

1. b) У найгіршому випадку потрібно або спуститися по найдовшій лівій гілці правого піддерева, або піднятися догори до першого “правого” повороту. В обох випадках довжина шляху обмежена висотою дерева, яка для збалансованого дерева є $O(\log N)$.

2. d) Видалення кореня з двома нащадками вимагає пошуку наступного елемента (що займає $O(\log N)$) і подальшого рекурсивного видалення цього наступного елемента. Це найскладніший сценарій.

3. d) У виродженому дереві (зв’язному списку) будь-яка операція, що потенційно вимагає обходу (вставка, пошук min/max), матиме складність $O(N)$.

4. a) Так. Останній елемент у post-order обході — це завжди корінь. Знайшовши його в in-order обході, ми можемо рекурсивно розділити задачу на ліве та праве піддерева. Комбінація pre-order та in-order також працює.

5. c) У найгіршому випадку (вироджене дерево), глибина рекурсії досягне N, і стек викликів функцій займе пам’ять, пропорційну N.

6. d) Класичний алгоритм видалення вузла з двома нащадками знаходить його in-order successor (найменший елемент у правому піддереві), переміщує його дані та видаляє вузол-наступник.

7. b) Кожна з N вставок в середньому займає $O(\log N)$ часу для випадкових даних (оскільки дерево, ймовірно, буде збалансованим). Таким чином, загальна очікувана складність побудови — $N \times O(\log N) = O(N\log N)$.

Завдання для самостійної роботи

1. Ітеративна реалізація. Напишіть ітеративні (без рекурсії) версії функцій `search` та `insert` для BST.
2. Обходи. Реалізуйте всі три види обходу в глибину (In-order, Pre-order, Post-order) та обхід в ширину.
3. Висота дерева. Напишіть функцію, що обчислює висоту бінарного дерева.
4. Валідація BST. Створіть функцію `isValidBST(node)`, яка перевіряє, чи є задане бінарне дерево коректним бінарним деревом пошуку.
5. k-тий найменший елемент. Напишіть функцію, що знаходить k-тий за величиною елемент у BST.
6. Найнижчий спільний предок (LCA). Реалізуйте функцію для знаходження найнижчого спільного предка для двох заданих вузлів у BST.
7. Діапазонний пошук. Напишіть функцію, що виводить всі ключі дерева, які знаходяться в заданому діапазоні `[min, max]`.
8. Побудова з масиву. Напишіть функцію, що будує збалансоване BST з відсортованого масиву.
9. Пошук попередника. Реалізуйте функцію для пошуку in-order predecessor для заданого вузла.
10. Видалення без рекурсії. Реалізуйте ітеративну версію функції `delete`.
11. Аналіз виродження. Проведіть експеримент: виміряйте час побудови та пошуку для BST з N елементів, вставлених у випадковому порядку та у відсортованому порядку. Побудуйте графіки.
12. Баланс. Напишіть функцію, що перевіряє, чи є дерево “майже збалансованим” (висоти лівого та правого піддерев для кожного вузла відрізняються не більше, ніж на 1).
13. Серіалізація. Напишіть функції для серіалізації (збереження у рядок) та десеріалізації (відновлення з рядка) бінарного дерева.
14. Дзеркальне відображення. Напишіть функцію, що створює дзеркальне відображення BST. Чи буде отримане дерево також BST?

15. Есе-порівняння. Напишіть есе, що порівнює BST та хеш-таблицю. Опишіть по 3 задачі, де кожна з цих структур є беззаперечно кращим вибором.

Збалансовані бінарні дерева пошуку

Ми переходимо до наступного логічного етапу у вивченні деревних структур. Розглянувши просте бінарне дерево пошуку, ми побачили його потужність, але й виявили його "ахіллесову п'яту". Цей розділ присвячений інженерним та математичним рішенням, які перетворюють BST на надійний та ефективний інструмент.

Проблема рівноваги

У попередньому розділі ми встановили, що бінарне дерево пошуку (BST) — це чудова структура, яка в середньому випадку забезпечує логарифмічний час $O(\log n)$ для операцій пошуку, вставки та видалення. Однак ми також дізналися про його фатальну ваду: у найгіршому випадку, коли дані вставляються у впорядкованому порядку, дерево вироджується у зв'язний список, і його продуктивність падає до лінійного часу $O(n)$.

Це ставить перед нами фундаментальне питання: як можна користуватися перевагами BST, але при цьому застрахувати себе від його найгіршого випадку? Як змусити дерево залишатися "кущистим" та "низьким", незалежно від порядку операцій?

Відповідь лежить у **самозбалансованих бінарних деревах пошуку**.

Це особливий клас BST, які мають вбудовані механізми для автоматичної підтримки балансу. Після кожної операції вставки або видалення, яка може порушити рівновагу, дерево виконує невеликі локальні "коригування", щоб відновити свою структуру і гарантувати, що його висота ніколи не буде значно більшою за $\log n$.

Основним інструментом для такої перебудови є операція **обертання (rotation)**. Це геніальний за своєю простотою локальний маневр, що змінює місцями вузол та одного з його нащадків, зберігаючи при цьому основну властивість бінарного дерева пошуку. Обертання виконується за константний час $O(1)$ і дозволяє ефективно зменшити висоту занадто довгої гілки.

Чотири філософії балансування

Не існує єдиного "ідеального" способу підтримувати баланс. Різні алгоритми пропонують різні компроміси між строгістю балансу та вартістю

його підтримки. У цьому розділі ми розглянемо чотири основні "філософії" балансування:

1. **AVL-дерево: "Перфекціоніст".** Це найстаріше самозбалансоване дерево. Воно дотримується дуже суворого правила: висоти лівого та правого піддерев для будь-якого вузла можуть відрізнятися не більше, ніж на одиницю. Це гарантує мінімально можливу висоту дерева, але може вимагати відносно частих операцій обертання для підтримки цього ідеального балансу.
2. **Червоно-чорне дерево (Red-Black Tree): "Прагматик".** Цей підхід використовує більш гнучкі правила балансування, засновані на "кольорах" вузлів (червоний або чорний). Дерево не є ідеально збалансованим, але гарантує, що найдовший шлях від кореня до листка не більше ніж удвічі довший за найкоротший. Це трохи збільшує висоту порівняно з AVL-деревом, але значно зменшує кількість обертань при вставці/видаленні, що робить його швидшим на практиці. Саме тому червоно-чорні дерева є промисловим стандартом і використовуються в стандартних бібліотеках багатьох мов.
3. **Splay-дерево (Розширюване дерево): "Опортуніст".** Це дерево взагалі не має жорстких правил балансу. Його магія полягає в іншому: щоразу, коли ми звертаємось до якогось вузла (для пошуку, вставки чи видалення), цей вузол за допомогою серії обертань переміщується в корінь дерева. Це призводить до того, що елементи, які використовуються часто, завжди знаходяться близько до вершини. Splay-дерево має чудову амортизаційну складність $O(\log n)$, хоча окрема операція може бути дорогою.
4. **Декартове дерево (Treap): "Гравець випадку".** Ця структура геніально поєднує дві ідеї. Вона є бінарним деревом пошуку за ключами і водночас купою (heap) за випадково згенерованими пріоритетами. Саме ця випадковість з високою ймовірністю гарантує, що дерево буде збалансованим. Декартові дерева часто значно простіші в реалізації, ніж AVL або червоно-чорні дерева.

У наступних підрозділах ми детально розберемо кожен з цих підходів, їхні алгоритми, переваги та недоліки.

AVL-дерево

Ми переходимо до першого і найбільш класичного представника родини самозбалансованих дерев. Цей розділ детально пояснить, як, застосовуючи строгі правила та елегантні операції “обертання”, AVL-дерево підтримує ідеальну рівновагу і гарантує високу продуктивність.

1. Теоретичні відомості

1.1. Ідея AVL-дерева: строгий контроль висоти

У попередньому розділі ми встановили, що головна проблема звичайного бінарного дерева пошуку (BST) — це його схильність до виродження у зв’язний список, що призводить до катастрофічного падіння продуктивності до $O(n)$. Щоб розв’язати цю проблему, нам потрібен механізм, який би автоматично підтримував дерево “кущистим” та низьким.

AVL-дерево, назване на честь його винахідників Г. М. Адельсона-Вельського та Є. М. Ландіса (1962), було першою винайденою структурою даних такого типу. Це самозбалансоване бінарне дерево пошуку, яке підтримує баланс за допомогою дуже строгого правила.

Властивість AVL-дерева (інваріант):

Для будь-якого вузла в дереві висоти його лівого та правого піддерев можуть відрізнятися не більше, ніж на 1.

Щоб відстежувати цю властивість, для кожного вузла вводиться поняття коефіцієнта балансу:

Коефіцієнт балансу = висота(праве_піддерево) – висота (ліве_піддерево)

У коректному AVL-дереві коефіцієнт балансу для кожного вузла може приймати лише три значення:

- -1: Ліве піддерево на 1 вище за праве.
- 0: Піддерева мають однакову висоту.
- +1: Праве піддерево на 1 вище за ліве.

Якщо після операції вставки або видалення коефіцієнт балансу якогось вузла стає -2 або +2, це означає, що баланс порушено, і дерево має виконати спеціальні операції для його відновлення. Для цього нам доведеться зберігати висоту для кожного вузла.

Структура вузла AVL-дерева:

```
Структура AVLNode:
```

```
    key       // Ключ (дані)
    left      // Вказівник на лівого нащадка
    right     // Вказівник на правого нащадка
    height    // Висота піддерева, коренем якого є цей вузол
```

Механізм балансування: обертання

Основним інструментом для відновлення балансу є обертання. Це локальна операція, що змінює структуру 2-3 вузлів, зберігаючи при цьому властивість бінарного дерева пошуку. Вона виконується за константний час $O(1)$. Існує два базових типи обертань.

Праве обертання (Right Rotation)

Праве обертання — це фундаментальний механізм балансування, який застосовується, коли дерево стає “ліво-важким”. Уявіть, що ви додали новий елемент у ліве піддерево лівого нащадка певного вузла. Цей вузол, який ми назвемо `y`, стає розбалансованим. Його коефіцієнт балансу стає -2, що сигналізує про порушення інваріанту AVL-дерева. Візуально це виглядає так, ніби ліва гілка дерева стала занадто довгою і “переважує” праву.

Мета операції

Головна мета правого обертання — “перегрупувати” вузли так, щоб відновити баланс висот, не порушивши при цьому основну властивість бінарного дерева пошуку:

`ліве < корінь < праве`.

Ми хочемо, щоб дерево стало більш “кущистим” і менш “високим”.

Дійові особи

Щоб зрозуміти процедуру, давайте дамо імена ключовим вузлам, які беруть участь в операції:

`y` — це розбалансований вузол (його коефіцієнт балансу -2). Він є “дідусем” у нашій структурі.

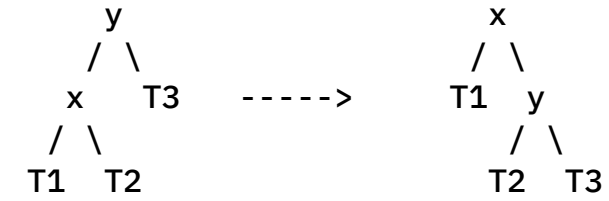
`x` — це лівий нащадок вузла `y`. Він є “батьком” і є коренем важчого, лівого піддерева.

`T2` — це праве піддерево вузла `x`. Це “внучате” піддерево, яке знаходиться між `x` та `y`. Це ключовий елемент, який буде переміщено.

Початкова конфігурація виглядає так: `y` є коренем, а його лівий нащадок — `x`.

Покрокова процедура обертання

Процес обертання можна порівняти з обережним маневром, що змінює ролі вузлів x та y. Вузол x “піднімається” вгору, стаючи новим коренем, а y “опускається” вниз, стаючи його правим нащадком.



Крок 1: Підготовка до маневру Ми ідентифікуємо вузол x (лівий нащадок y) та його праве піддерево T2. Це піддерево є критично важливим, оскільки воно містить елементи, які більші за x, але менші за y.

Крок 2: “Усиновлення” піддерева Вузол y більше не може мати x своїм лівим нащадком, оскільки x сам стане коренем. Отже, лівий вказівник y звільняється. Ідеальне місце для нього займає піддерево T2. Ми робимо T2 новим лівим нащадком вузла y. Це коректно, оскільки всі елементи в T2 менші за y.

Крок 3: “Просування” вузла x Тепер, коли x звільнився від свого правого нащадка (T2), він “піднімається” на місце y. Ми змінюємо правий вказівник x так, щоб він вказував на y. Це також коректно, оскільки y більший за x.

Крок 4: Завершення та оновлення Після цих маніпуляцій з вказівниками структура змінюється. Новим коренем цього фрагмента дерева стає x. Його лівим нащадком залишається його початкове ліве піддерево, а його новим правим нащадком стає y. Вузол y, у свою чергу, має своїм лівим нащадком T2, а правим — своє початкове праве піддерево.

Останній, але не менш важливий крок — це оновлення висот. Оскільки структура змінилася, висоти вузлів x та y (і, можливо, їхніх предків) також змінилися. Ми повинні переобчислити їх, рухаючись знизу вгору. Спочатку оновлюється висота нового нащадка (y), а потім — нового кореня (x).

Результат: Властивість бінарного дерева пошуку повністю збережена. Висота піддерева, яке було занадто “важким”, зменшилася, і баланс вузла y (і x) тепер становить 0 або -1, що відповідає інваріанту AVL-дерева.

Псевдокод правого обертання:

```
ФУНКЦІЯ rightRotate(y):
    x = y.left
    T2 = x.right
```

```
    // Виконуємо обертання
    x.right = y
    y.left = T2

    // Оновлюємо висоти (спочатку для y, потім для x, бо висота x зал
ежить від нової висоти y)
    y.height = 1 + max(height(y.left), height(y.right))
    x.height = 1 + max(height(x.left), height(x.right))

    // Повертаємо новий корінь піддерева
    ПОВЕРНУТИ x
КІНЕЦЬ ФУНКЦІЇ
```

Ліве обертання (Left Rotation)

Ліве обертання є дзеркальним до правого і застосовується, коли дерево стає “право-важким”. Така ситуація виникає після операції вставки або видалення у праве піддерево правого нащадка певного вузла. Цей вузол, який ми назвемо `x`, стає розбалансованим, і його коефіцієнт балансу дорівнює +2. Візуально це виглядає так, ніби права гілка дерева стала занадто довгою і “переважує” ліву.

Мета операції

Як і у випадку з правим обертанням, мета — відновити баланс висот, зберігши при цьому властивість бінарного дерева пошуку (**`ліве < корінь < праве`**). Ми хочемо реорганізувати вузли, щоб “підтягнути” коротшу ліву гілку і зробити дерево більш збалансованим та менш високим.

Дійові особи

Для розуміння процедури визначимо ключові вузли, що беруть участь в операції:

`x` — це розбалансований вузол (його коефіцієнт балансу +2). Він є “дідусем”.

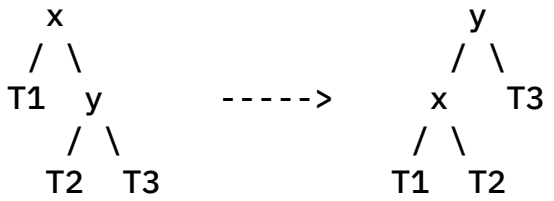
`y` — це правий нащадок вузла `x`. Він є “батьком” і коренем важчого, правого піддерева.

`T2` — це ліве піддерево вузла `y`. Це “внучате” піддерево, яке знаходиться між `x` та `y` і буде переміщене.

Початкова конфігурація: `x` є коренем, а його правий нащадок — `y`.

Покрокова процедура обертання

Процес є дзеркальним до правого обертання. Тепер вузол `y` “піднімається” вгору, стаючи новим коренем, а `x` “опускається” вниз, стаючи його лівим нащадком.



Крок 1: Підготовка до маневру Ми ідентифікуємо вузол `y` (правий нащадок `x`) та його ліве піддерево `T2`. Це піддерево є ключовим, оскільки містить елементи, які менші за `y`, але більші за `x`.

Крок 2: “Усиновлення” піддерева Вузол `x` більше не може мати `y` своїм правим нащадком, оскільки `y` сам стане коренем. Отже, правий вказівник `x` звільняється. Ідеальне місце для нього займає піддерево `T2`. Ми робимо `T2` новим правим нащадком вузла `x`. Це коректно, оскільки всі елементи в `T2` більші за `x`.

Крок 3: “Просування” вузла `y` Тепер, коли `y` звільнився від свого лівого нащадка (`T2`), він “піднімається” на місце `x`. Ми змінюємо лівий вказівник `y` так, щоб він вказував на `x`. Це також коректно, оскільки `x` менший за `y`.

Крок 4: Завершення та оновлення Після цих маніпуляцій структура змінюється. Новим коренем цього фрагмента дерева стає `y`. Його правим нащадком залишається його початкове праве піддерево, а його новим лівим нащадком стає `x`. Вузол `x`, у свою чергу, має своїм правим нащадком `T2`, а лівим — своє початкове ліве піддерево.

Останній крок — оновлення висот. Ми переобчислюємо висоту для змінених вузлів, рухаючись знизу вгору: спочатку для `x`, а потім для `y`.

Результат: Властивість бінарного дерева пошуку повністю збережена. Висота піддерева, яке було занадто “важким”, зменшилася, і баланс вузла `x` (і `y`) тепер знаходиться в межах `[-1, 1]`, що відповідає інваріанту AVL-дерева.

Псевдокод лівого обертання:

```
ФУНКЦІЯ leftRotate(x):
    y = x.right
    T2 = y.left

    // Виконуємо обертання
    y.left = x
    x.right = T2

    // Оновлюємо висоти (спочатку для x, потім для y)
    x.height = 1 + max(height(x.left), height(x.right))
```

```
    y.height = 1 + max(height(y.left), height(y.right))

    // Повертаємо новий корінь піддерева
    ПОВЕРНУТИ y
КІНЕЦЬ ФУНКЦІЇ
```

1.2. Сценарії порушення балансу та їх виправлення

Після вставки або видалення вузла, ми піднімаємось по дереву до кореня, оновлюючи висоти та перевіряючи коефіцієнт балансу. Якщо ми знаходимо вузол **z**, де баланс порушено, існує 4 можливих сценарії.

1. Ліво-Лівий випадок (Left-Left Case): Вставка відбулась у ліве піддерево лівого нащадка вузла **z**. Коефіцієнт балансу **z** дорівнює -2.
    - Рішення: Одне праве обертання навколо **z**.
2. Право-Правий випадок (Right-Right Case): Вставка відбулась у праве піддерево правого нащадка **z**. Коефіцієнт балансу **z** дорівнює +2.
    - Рішення: Одне ліве обертання навколо **z**.
3. Ліво-Правий випадок (Left-Right Case): Вставка відбулась у праве піддерево лівого нащадка **z**.
    - Рішення: Подвійне обертання: спочатку ліве обертання навколо лівого нащадка **z**, що зводить задачу до Ліво-Лівого випадку, а потім праве обертання навколо **z**.
4. Право-Лівий випадок (Right-Left Case): Вставка відбулась у ліве піддерево правого нащадка **z**.
    - Рішення: Подвійне обертання: спочатку праве обертання навколо правого нащадка **z**, що зводить задачу до Право-Правого випадку, а потім ліве обертання навколо **z**.

1.3. Операції та їх псевдокод

Пошук

Пошук в AVL-дереві нічим не відрізняється від пошуку в звичайному BST, оскільки обертання зберігають властивість впорядкованості. *Складність:* $O(\log n)$

Вставка

Вставка — це звичайна вставка в BST з подальшим підйомом по дереву, оновленням висот та виконанням балансуючих обертань у разі потреби.

```
ФУНКЦІЯ Insert(node, key):
    // 1. Стандартна вставка в BST
```

```
    ЯКЩО node == NULL ТО ПОВЕРНУТИ Створити_Вузол(key) КІНЕЦЬ ЯКЩО

    ЯКЩО key < node.key ТО
        node.left = Insert(node.left, key)
    ІНАКШЕ ЯКЩО key > node.key ТО
        node.right = Insert(node.right, key)
    ІНАКШЕ // Дублікати не дозволені
        ПОВЕРНУТИ node
    КІНЕЦЬ ЯКЩО

    // 2. Оновлення висоти поточного вузла
    node.height = 1 + max(height(node.left), height(node.right))

    // 3. Обчислення коефіцієнта балансу
    balance = getBalance(node)

    // 4. Перевірка на розбалансування і виконання обертань
    // Ліво-Лівий випадок
    ЯКЩО balance < -1 І key < node.left.key ТО
        ПОВЕРНУТИ rightRotate(node)
    КІНЕЦЬ ЯКЩО
    // Право-Правий випадок
    ЯКЩО balance > 1 І key > node.right.key ТО
        ПОВЕРНУТИ leftRotate(node)
    КІНЕЦЬ ЯКЩО
    // Ліво-Правий випадок
    ЯКЩО balance < -1 І key > node.left.key ТО
        node.left = leftRotate(node.left)
        ПОВЕРНУТИ rightRotate(node)
    КІНЕЦЬ ЯКЩО
    // Право-Лівий випадок
    ЯКЩО balance > 1 І key < node.right.key ТО
        node.right = rightRotate(node.right)
        ПОВЕРНУТИ leftRotate(node)
    КІНЕЦЬ ЯКЩО

    // Якщо баланс не порушено, повертаємо незмінений вузол
    ПОВЕРНУТИ node
КІНЕЦЬ ФУНКЦІЇ
```

Складність: $O(\log n)$, оскільки і спуск, і підйом з балансуванням займають логарифмічний час.

Видалення

Видалення також аналогічне до вставки: спершу виконується стандартне видалення з BST, а потім, на шляху повернення рекурсії, виконується балансування. *Складність:* $O(\log n)$

Обходи дерева

Алгоритми обходу для AVL-дерева є абсолютно ідентичними до алгоритмів для звичайного бінарного дерева, оскільки вони залежать лише від структури, а не від балансу. *Складність:* $O(n)$

2. Область застосування

AVL-дерева використовуються в системах, де потрібен гарантований логарифмічний час пошуку, і де операції читання (пошуку) відбуваються значно частіше, ніж операції запису (вставка/видалення).

- Бази даних: У деяких реалізаціях індексів, де швидкість пошуку є найвищим пріоритетом.
- Словники та множини: У бібліотеках, що надають гарантовану продуктивність у найгіршому випадку.
- Мережеві маршрутизатори: Для зберігання таблиць маршрутизації.

3. Приклади застосування

- Символьні таблиці в компіляторах: Хоча частіше використовуються червоно-чорні дерева, AVL-дерева також можуть застосовуватися для ефективного управління ідентифікаторами.
- Системи, що вимагають частих діапазонних запитів: Завдяки впорядкованості, AVL-дерева дозволяють швидко знаходити всі елементи в заданому діапазоні.

Аналіз складності операцій в AVL-дереві

Фундаментальна властивість AVL-дерева, яка визначає всю його продуктивність, — це гарантована логарифмічна висота. Через строгий контроль балансу, висота h дерева з N вузлами завжди є $O(\log N)$. Це усуває можливість виродження дерева у зв’язний список, а отже, і катастрофічний найгірший випадок $O(N)$, притаманний звичайним BST.

Пошук

- Найгірший випадок: $O(\log N)$ У найгіршому випадку нам доведеться спуститися від кореня до найглибшого листка. Оскільки висота дерева гарантовано не перевищує $O(\log N)$, кількість порівнянь також обмежена цією величиною.

– Середній випадок: $O(\log N)$ Середня довжина шляху у збалансованому дереві також є логарифмічною.

Вставка

– Найгірший випадок: $O(\log N)$ Процес складається з двох частин:

7. Спуск до місця вставки, що займає $O(\log N)$.
8. Підйом до кореня для оновлення коефіцієнтів балансу та, за потреби, виконання обертань. Важливо, що для операції вставки потрібно не більше одного балансуючого обертання (простого чи подвійного), яке виконується за $O(1)$. Загальна складність визначається спуском/підйомом по дереву.

– Середній випадок: $O(\log N)$ Аналогічно до найгіршого випадку, оскільки структура дерева завжди підтримується збалансованою.

Видалення

– Найгірший випадок: $O(\log N)$ Процес схожий на вставку, але має важливий нюанс:

9. Пошук вузла та його наступника (якщо потрібно) займає $O(\log N)$.
10. Підйом до кореня для балансування. На відміну від вставки, видалення може порушити баланс на декількох рівнях, що потенційно може вимагати до $O(\log N)$ обертань на шляху до кореня. Оскільки кожне обертання коштує $O(1)$, загальна вартість балансування — $O(\log N)$. Сумарна складність операції також становить $O(\log N)$.

– Середній випадок: $O(\log N)$ Залишається логарифмічним через гарантовану збалансованість дерева.

Обходи дерева

– Найгірший та середній випадок: $O(N)$ Будь-який повний обхід дерева (симетричний, прямий, зворотний) вимагає відвідування кожного з N вузлів рівно один раз. Тому часова складність завжди лінійна і залежить від кількості вузлів, а не від висоти.

Гаразд. Ось детальний аналіз складності для всіх операцій в AVL-дереві, що враховує середній та найгірший випадки.

Переваги та недоліки

Переваги: Головна перевага AVL-дерева — це його строгий баланс. Воно гарантує, що висота дерева з N вузлами ніколи не перевищить $1.44\log_2 N$, що робить його дуже “низьким” та “широким”. Це забезпечує надзвичайно швидкий пошук, швидший, ніж у більш “розслаблених” дерев типу червоно-чорних. Всі основні операції мають гарантовану складність $O(\log n)$ у найгіршому випадку.

Недоліки: Ціна за строгий баланс — це витрати на його підтримку. Вимога, щоб висоти піддерев відрізнялися не більше ніж на 1, може призводити до відносно частих операцій обертання при вставці та видаленні. Кожна така операція, хоч і має складність $O(1)$, вимагає виконання певних дій. Через це для задач, де операції запису є дуже частими, червоно-чорні дерева, які роблять менше обертань, можуть виявитись на практиці швидшими. Реалізація AVL-дерева також є досить складною через необхідність відстежувати висоти та коректно реалізовувати всі чотири сценарії обертань.

Контрольні запитання

1. Сформулюйте інваріант AVL-дерева. Що таке коефіцієнт балансу?
2. Яка операція використовується для відновлення балансу в AVL-дереві? Яка її складність?
3. Опишіть Ліво-Правий випадок розбалансування і які обертання потрібні для його виправлення.
4. Чому операція пошуку в AVL-дереві не вимагає жодних змін порівняно зі звичайним BST?
5. Порівняйте AVL-дерево та червоно-чорне дерево. У якому випадку AVL-дерево буде кращим вибором?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація AVL-дерева з операціями самобалансування.

2. Мета

Практичне засвоєння механізмів самобалансування бінарних дерев пошуку. Реалізація операцій обертання та їх застосування для підтримки інваріанту AVL-дерева.

3. Завдання

Створити програму, що реалізує AVL-дерево для зберігання цілих чисел та надає користувачеві меню для виконання операцій вставки та пошуку, а також виведення дерева.

4. Опис

Програма має складатися з класу/структури `AVLNode` (з полями `key`, `left`, `right`, `height`) та основного класу `AVLTree`. Основна складність полягає в коректній реалізації методу вставки, який включає логіку балансування.

5. Порядок виконання

1. Створіть клас `AVLNode`.
2. Реалізуйте допоміжні функції: Напишіть функції `height(node)` (повертає висоту вузла), `getBalance(node)` (обчислює коефіцієнт балансу).
3. Реалізуйте обертання: Напишіть функції `rightRotate(node)` та `leftRotate(node)`.
4. Реалізуйте `insert(key)`: Це головна функція. Вона має бути рекурсивною. Після стандартної вставки BST, на етапі повернення з рекурсії, вона повинна оновлювати висоту, обчислювати баланс і, якщо потрібно, виконувати один з чотирьох сценаріїв балансування.
5. Реалізуйте обхід: Додайте метод для симетричного обходу, щоб перевіряти коректність роботи.
6. Протестуйте: Створіть у `main` меню. Перевірте роботу, вставляючи послідовності, які викликають усі 4 типи обертань (напр., послідовна вставка 10, 20, 30 для простого обертання).

6. Результат

Додаток, що реалізує самозбалансоване AVL-дерево.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією всіх необхідних методів.
2. Продемонструвати роботу додатку, показавши, як дерево балансується після вставок, що порушують рівновагу.
3. Бути готовим відповісти на питання:
    - Пояснити, як працює будь-яке з обертань у вашому коді.

- Як ваша програма визначає, який саме тип обертання (з 4 можливих) потрібно застосувати?
- Чому висоту вузлів потрібно оновлювати саме в тому порядку, як це зроблено у функціях обертання?

Тести

1. Яке значення НЕ може приймати коефіцієнт балансу у вузлі коректного AVL-дерева?

a) -1

b) 0

c) 1

d) 2

2. Яка часова складність операції вставки в AVL-дерево у найгіршому випадку?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(N \log N)$

3. Яка операція є базовою для механізму балансування в AVL-дереві?

a) Видалення вузла.

b) Перефарбовування вузла.

c) Обертання.

d) Злиття піддерев.

4. Якщо вставка нового елемента відбулась у ліве піддерево правого нащадка (“Право-Лівий випадок”), яке балансування потрібне?

a) Одне ліве обертання.

b) Одне праве обертання.

c) Праве обертання, а потім ліве.

d) Ліве обертання, а потім праве.

5. Основна перевага AVL-дерева над звичайним BST полягає у:

a) Меншому використанні пам’яті.

b) Гарантованій логарифмічній складності операцій у найгіршому випадку.

c) Простішій реалізації.

d) Швидшій вставці у відсортованому порядку.

Відповіді на прості тести: 1-d, 2-b, 3-c, 4-c, 5-b.

Тести підвищеної складності

1. Скільки максимум операцій обертання (простих чи подвійних) може знадобитися для відновлення балансу після однієї операції вставки в AVL-дерево?

a) 1

b) 2

c) log$N$

d) N

2. Чим відрізняється балансування після операції видалення порівняно з балансуванням після вставки в AVL-дереві?

a) Нічим, логіка абсолютно ідентична.

b) Після видалення може знадобитися виконати обертання на кожному рівні дерева на шляху до кореня.

c) Після видалення використовуються інші типи обертань.

d) Видалення ніколи не порушує баланс.

3. Яке твердження про висоту h AVL-дерева з N вузлами є найбільш точним?

a) $h \approx \log_2 N$

b) $h < 1.44 < log_2 N$

c) $h = N - 1$ у найгіршому випадку.

d) Висота завжди є цілим числом.

4. Ви реалізуєте AVL-дерево, але замість зберігання повної висоти в кожному вузлі, ви вирішили зберігати лише коефіцієнт балансу (-1, 0, 1). Як це вплине на алгоритм?

a) Ніяк, це еквівалентні підходи.

b) Це неможливо, оскільки для обертань потрібна точна висота.

c) Це можливо, але значно ускладнить логіку оновлення балансу після операцій, оскільки зміну висоти доведеться виводити опосередковано.

d) Це значно прискорить роботу, оскільки коефіцієнт балансу займає менше місця.

5. У порівнянні з червоно-чорним деревом, AVL-дерево зазвичай виконує:

a) Більше обертань при модифікації, але забезпечує швидший пошук.

b) Менше обертань при модифікації, але має повільніший пошук.

c) Однакове число обертань та однакову швидкість пошуку.

d) Більше операцій порівняння при пошуку.

6. Якщо в AVL-дереві виконати симетричний обхід (in-order), а потім з отриманого відсортованого масиву побудувати нове AVL-дерево, чи буде воно ідентичним початковому?

a) Так, завжди.

b) Ні, ніколи.

c) Лише якщо початкове дерево було повним.

d) Не обов’язково; може існувати декілька різних AVL-дерев для одного і того ж набору ключів.

7. Який випадок балансування не вимагає подвійного обертання?

a) Вставка у праве піддерево лівого нащадка (Left-Right).

b) Вставка у ліве піддерево правого нащадка (Right-Left).

c) Вставка у ліве піддерево лівого нащадка (Left-Left).

d) Усі випадки можуть вимагати подвійного обертання.

8. Складність операції `getBalance(node)` в реалізації, де кожен вузол зберігає свою висоту, дорівнює:

a) $O(1)$

b) $O(\log N)$

c) $O(H)$, де H - висота вузла

d) $O(N)$

Відповіді на тестові питання підвищеної складності

1. *a)* Після вставки достатньо виконати одне балансування (просте або подвійне обертання) на першому розбалансованому вузлі на шляху до кореня. Це автоматично відновить інваріант для всього дерева.

2. *b)* На відміну від вставки, видалення може зменшити висоту піддерева, що потенційно може порушити баланс на всіх рівнях вище. Тому може знадобитися до $O(\log N)$ обертань.

3. *b)* Це класична теоретична оцінка. Через строгий інваріант висота AVL-дерева дуже близька до ідеальної висоти повного бінарного дерева, і доведено, що вона ніколи не перевищує її більше, ніж на 44%.

4. *c)* Це можлива реалізація, але вона складніша. При вставці/видаленні потрібно не просто оновити висоту, а проаналізувати, як змінився коефіцієнт балансу, і на основі цього виводити, чи змінилася висота всього піддерева.

5. *a)* AVL-дерево є більш строго збалансованим, тому його висота в середньому менша, що робить пошук трохи швидшим. Однак підтримка цього строгого балансу вимагає частіших і, можливо, складніших обертань при вставках/видаленнях.

6. *d)* Для одного набору ключів може існувати багато різних конфігурацій AVL-дерев. Наприклад, для ключів {1, 2, 3} і {2, 1, 3} будуть побудовані різні за структурою, але однаково коректні AVL-дерева.

7. *c)* Випадки, коли розбалансування відбувається “по прямій лінії” (LL або RR), виправляються одним відповідним обертанням. “Злами” (LR або RL) вимагають подвійного обертання.

8. *a)* Оскільки ми зберігаємо вже обчислену висоту в кожному вузлі, для знаходження коефіцієнта балансу достатньо взяти висоти лівого та правого нащадків і виконати одну операцію віднімання. Це займає константний час.

Завдання для самостійної роботи

2. Реалізація. Повністю реалізуйте клас AVL-дерева з операціями вставки та пошуку.
3. Реалізація видалення. Додайте до вашого класу метод `delete`, що підтримує балансування після видалення.

4. Ручне трасування. Вручну намалюйте, як буде змінюватися AVL-дерево при послідовній вставці ключів: 10, 20, 30, 40, 50. Покажіть усі необхідні обертання.
5. Аналіз обертань. Доведіть, що одне подвійне обертання (напр., Ліво-Праве) відновлює баланс для всіх вузлів у піддереві.
6. Ітеративна вставка. Спробуйте розробити ітеративну (нерекурсивну) версію методу вставки в AVL-дерево.
7. Порівняння з BST. Проведіть емпіричний аналіз: порівняйте час побудови та пошуку для AVL-дерева та звичайного BST на двох наборах даних: випадковому та відсортованому.
8. Діапазонний запит. Реалізуйте для вашого AVL-дерева функцію, що знаходить усі елементи в заданому діапазоні `[min, max]`.
9. Підрахунок обертань. Модифікуйте вашу реалізацію так, щоб вона підраховувала загальну кількість виконаних простих та подвійних обертань.
10. Есе-порівняння. Напишіть есе, що порівнює AVL-дерево та червоно-чорне дерево. У яких випадках кожне з них є кращим вибором?
11. Перевірка на AVL. Напишіть функцію, яка перевіряє, чи є задане бінарне дерево коректним AVL-деревом.

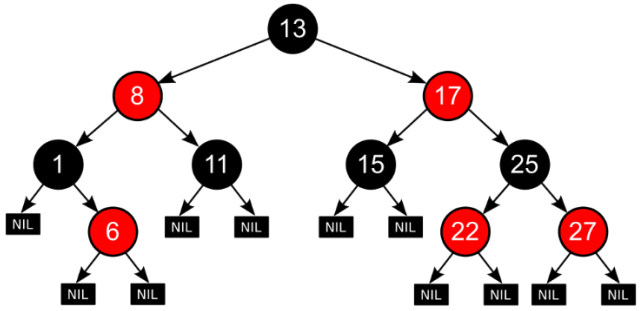
Червоно-чорне дерево

Розглянувши AVL-дерево, яке є “перфекціоністом” у підтримці балансу, ми переходимо до його більш гнучкого та прагматичного “колеги”. Червоно-чорне дерево — це одна з найпоширеніших самозбалансованих структур, яка лежить в основі багатьох стандартних бібліотек та систем, і розуміння її роботи є ключовим для сучасного інженера.

1. Теоретичні відомості

1.1. Ідея червоно-чорного дерева: баланс через гнучкі правила

AVL-дерево гарантує майже ідеальну збалансованість, але розплачується за це потенційно великою кількістю обертань при модифікації. Виникає питання: чи можна трохи “послабити” вимоги до балансу, щоб зменшити вартість його підтримки, але все ще гарантувати логарифмічну складність? Відповідь — так, і це філософія червоно-чорного дерева (Red-Black Tree, RBT).



Замість того, щоб зберігати точну висоту в кожному вузлі, RBT використовує один додатковий біт інформації — колір вузла (червоний або чорний). Балансування досягається не за рахунок строгого контролю висот, а за рахунок дотримання набору з п’яти простих правил, що стосуються цих кольорів.

Структура вузла червоно-чорного дерева:

```
Структура RBTNode:
    key        // Ключ (дані)
    color      // Колір (ЧЕРВОНИЙ або ЧОРНИЙ)
    left       // Вказівник на лівого нащадка
    right      // Вказівник на правого нащадка
    parent     // Вказівник на батьківський вузол (дуже корисний для р
еалізації)
```

Кількість чорних вузлів на шляху від вузла x (не рахуючи сам вузол) до листа будемо називати чорною висотою вузла (black-height) і позначати як bh (x).

1.2. П’ять правил червоно-чорного дерева

Будь-яке коректне червоно-чорне дерево повинно завжди задовольняти наступним п’яти інваріантам:

1. Правило кольору: Кожен вузол є або червоним, або чорним.
2. Правило кореня: Корінь дерева завжди чорний.
3. Правило листків: Усі листки (кінцеві вузли `NULL`) є чорними. (Це уявні, “фантомні” вузли, що спрощують алгоритми).
4. Правило червоних вузлів: Якщо вузол червоний, то обидва його нащадки обов’язково чорні. Це означає, що не може бути двох червоних вузлів поспіль на одному шляху від кореня до листка.
5. Правило чорної висоти: Для будь-якого вузла, всі шляхи від нього до його нащадків-листків містять однакову кількість чорних вузлів.

Саме правила 4 і 5 є серцем механізму балансування. Вони гарантують, що найдовший можливий шлях у дереві (де чергуються чорні та червоні вузли)

не може бути довшим, ніж удвічі за найкоротший можливий шлях (що складається лише з чорних вузлів). Це, у свою чергу, забезпечує висоту дерева на рівні $O(\log n)$ і гарантує логарифмічну складність усіх операцій.

1.3. Операції та механізми балансування

Як і в AVL-дереві, пошук та обходи реалізуються так само, як у звичайному BST. Вся складність полягає в операціях вставки та видалення, які можуть порушити червоно-чорні правила. Для відновлення балансу RBT використовує два інструменти:

- Перефарбовування (Recoloring): Зміна кольору вузла та його сусідів. Це дуже швидка операція.
- Обертання (Rotation): Ті самі ліві та праві обертання, що ми розглядали для AVL-дерев.

Вставка вузла у червоно-чорне дерево

Ми вже знаємо, що червоно-чорне дерево — це бінарне дерево пошуку, яке підтримує баланс за допомогою п’яти правил, що стосуються кольорів вузлів. Будь-яка операція модифікації, як-от вставка, несе ризик порушення цих правил.

Перший крок вставки — це завжди стандартна вставка, як у звичайному BST. Ми знаходимо відповідне місце для нового вузла і приєднуємо його. Але якого кольору він має бути? У нас є два варіанти:

1. Зафарбувати новий вузол у чорний: Це погана ідея. Якщо ми додамо чорний вузол, ми майже гарантовано порушимо правило №5 (правило чорної висоти). Будь-який шлях, що пройде через наш новий вузол, отримає додатковий чорний вузол, і чорна висота дерева стане некоректною. Виправлення цього порушення є дуже складною операцією.
2. Зафарбувати новий вузол у червоний: Це значно краща стратегія. Вставка червоного вузла ніколи не порушує правило чорної висоти, оскільки червоні вузли на неї не впливають. Однак, вона може порушити правило №4, якщо батько нового вузла також виявиться червоним (ситуація “два червоні вузли поспіль”).

Саме тому новий вузол при вставці в червоно-чорне дерево завжди зафарбовується у червоний колір. Це створює лише одну потенційну проблему, яку значно легше виправити, ніж порушення чорної висоти.

Загальний алгоритм вставки:

1. Виконати стандартну вставку вузла, як у звичайному BST.
2. Зафарбувати новий вузол у червоний колір.
3. Якщо виникла ситуація “червоний батько — червоний син”, викликати спеціальну процедуру відновлення балансу **`fix_insertion()`**.

1.2. Механізм відновлення: аналіз випадків

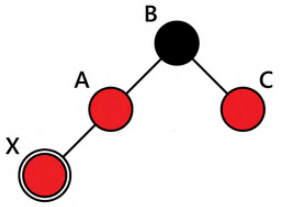
Процедура відновлення **`fix_insertion`** — це серце операції. Вона аналізує кольори “родичів” нашого нового вузла **`x`** (його батька **`parent`**, дідуся **`grandparent`** та “дядька” **`uncle`** — брата батька) і виконує перефарбовування та обертання, щоб усунути конфлікт “червоний-червоний”.

Розглянемо всі можливі випадки, припускаючи, що **`x.parent`** є лівим нащадком **`x.grandparent`**. (Випадки, коли батько є правим нащадком, є дзеркально симетричними).

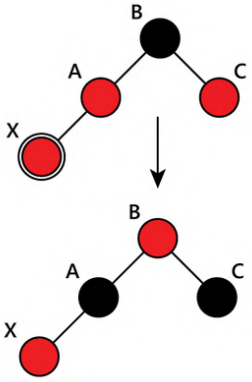
Детальний розбір випадків:

Випадок 1: “Дядько” `uncle` — червоний.

- Ситуація: Ми маємо червоного батька і червоного дядька. Це означає, що їхній спільний батько (дідусь) точно чорний.



- Аналогія: Це схоже на вирішення локального конфлікту шляхом “підняття” проблеми на вищий рівень управління.
- Рішення: Перефарбовування. Ми перефарбовуємо батька і дядька в чорний колір, а дідуся — в червоний.

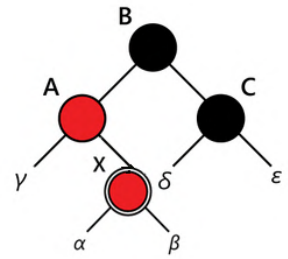


- Результат: Правило “два червоні поспіль” на поточному рівні вирішено. Однак, тепер дідусь став червоним, і якщо його батько теж був червоним, конфлікт просто перемістився на два рівні вгору по дереву. Тому ми переміщуємо наш вказівник `x` на дідуся і повторюємо перевірку в наступній ітерації циклу.

Випадок 2: “Дядько” `uncle` — чорний (або NULL).

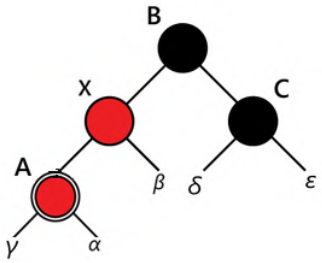
Ця ситуація складніша і вимагає зміни структури дерева за допомогою обертань. Вона, в свою чергу, ділиться на два під-випадки, залежно від того, чи утворюють дідусь, батько і син “лінію” чи “трикутник”.

- Випадок 2а (конфігурація “трикутник” / Left-Right Case):
  - Ситуація: `x` є правим нащадком свого батька, а батько — лівим нащадком дідуся. Візуально вони утворюють зигзаг або трикутник.



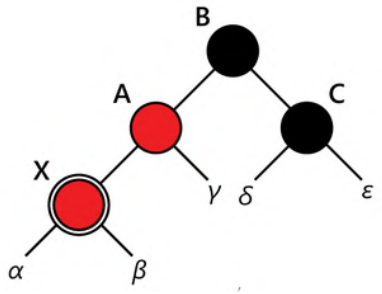
  - Рішення: Одним обертанням цю ситуацію не виправити. Ми спершу зводимо її до простішої конфігурації “лінія”. Для цього ми виконуємо ліве обертання навколо батька `x.parent`.

    Після цього `x` та його колишній батько міняються місцями, і `x` стає лівим нащадком.

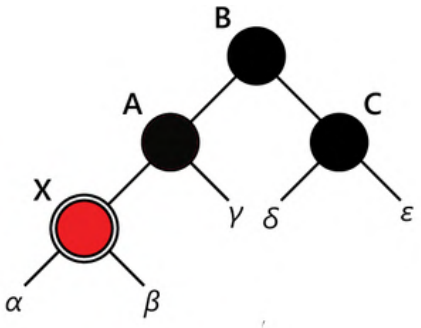


  - Результат: Конфігурація “трикутник” перетворилась на конфігурацію “лінія” (випадок 2б).
- Випадок 2б (конфігурація “лінія” / Left-Left Case):

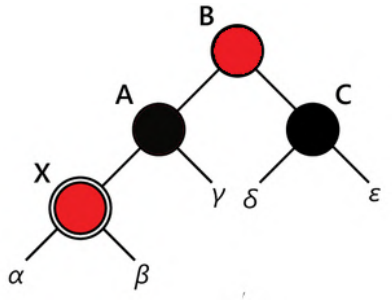
- Ситуація: `x` є лівим нащадком свого батька, а батько — лівим нащадком дідуся. Візуально вони утворюють пряму лінію.



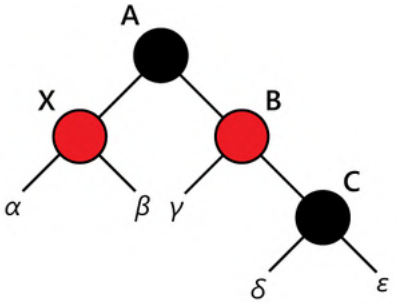
- Рішення: Це фінальний випадок, який повністю вирішує проблему. Ми виконуємо такі дії:
    1. Перефарбовуємо батька `x.parent` в чорний.



    2. Перефарбовуємо дідуся `grandparent` в червоний.



    3. Виконуємо праве обертання навколо дідуся.



- Результат: Порушення правила “два червоні поспіль” усунуто. Більше того, оскільки ми додали один чорний

вузол (колишній батько) і один червоний (колишній дідусь) до гілки, що проходить через дядька (який був чорним), правило чорної висоти для всього дерева залишається коректним. На цьому цикл відновлення завершується.

Дзеркальні випадки, коли батько є правим нащадком, обробляються аналогічно, просто замінюючи “ліве” на “праве” і навпаки.

Псевдокод основної логіки відновлення:

```
// Вхід: x - щойно вставлений червоний вузол
ФУНКЦІЯ fix_insertion(x):
    // Цикл триває, доки батько 'x' є червоним. Якщо батько чорний,
    // жодне правило не порушено, і нічого робити не потрібно.
    ПОКИ x.parent != NULL І x.parent.color == ЧЕРВОНИЙ:
        grandparent = x.parent.parent

        // --- Розглядаємо випадок, коли батько 'x' є ЛІВИМ нащадком
---
        ЯКЩО x.parent == grandparent.left ТО
            uncle = grandparent.right // "Дядько" x

            // ---> Випадок 1: "Дядько" червоний
            ЯКЩО uncle != NULL І uncle.color == ЧЕРВОНИЙ ТО
                x.parent.color = ЧОРНИЙ
                uncle.color = ЧОРНИЙ
                grandparent.color = ЧЕРВОНИЙ
                // Переміщуємо проблему на два рівні вгору
                x = grandparent
            // ---> Випадок 2: "Дядько" чорний (або NULL, що теж чорн
ий)
            ІНАКШЕ:
                // Випадок 2а (конфігурація "трикутник"): 'x' є прави
м нащадком
                ЯКЩО x == x.parent.right ТО
                    x = x.parent
                    leftRotate(x) // Зводимо до випадку "лінії"
                КІНЕЦЬ ЯКЩО

                // Випадок 2б (конфігурація "лінія"): 'x' є лівим нащ
адком
                x.parent.color = ЧОРНИЙ
                grandparent.color = ЧЕРВОНИЙ
                rightRotate(grandparent)
            КІНЕЦЬ ЯКЩО
```

```
        // --- Розглядаємо випадок, коли батько 'x' є ПРАВИМ нащадком
(дзеркально)
        ІНАКШЕ:
            // ... аналогічна логіка з leftRotate та rightRotate ...
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ

    // Правило №2: Корінь дерева завжди має бути чорним.
    // Навіть якщо в процесі перефарбовувань він став червоним,
    // ми примусово робимо його чорним в кінці.
    root.color = ЧОРНИЙ
КІНЕЦЬ ФУНКЦІЇ
```

Таким чином, за допомогою невеликої кількості перефарбовувань та максимум двох обертань, алгоритм вставки гарантовано відновлює всі інваріанти червоно-чорного дерева, зберігаючи його збалансованість та логарифмічну висоту.

Видалення вузла з червоно-чорного дерева

Якщо вставка нового (червоного) вузла могла порушити лише одне правило (два червоні вузли поспіль), то видалення, особливо видалення чорного вузла, є значно більш деструктивною операцією. Коли ми видаляємо чорний вузол, ми, по суті, створюємо “чорну діру” на всіх шляхах, що проходили через нього. Це порушує найважливіше правило №5: *всі шляхи від будь-якого вузла до його листків мають містити однакову кількість чорних вузлів*. Шляхи, що проходили через видалений вузол, раптом стають на один чорний вузол коротшими за інші, і баланс всього дерева руйнується.

Саме для вирішення цієї проблеми “нестачі чорного” і була розроблена процедура відновлення балансу після видалення. Наша мета — позбутися цієї “нестачі”, переміщуючи її вгору по дереву або компенсуючи за допомогою перефарбовувань та обертань.

Але спочатку розглянемо саме «видалення». При видаленні вершини можуть виникнути три випадки в залежності від кількості дітей:

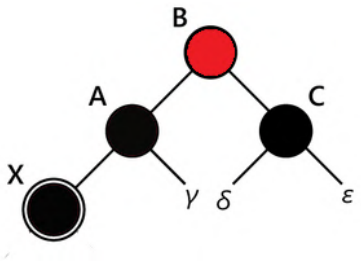
1. Якщо вершина не має дітей

2. Якщо у неї лише одна дитина

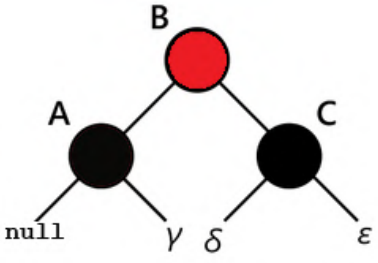
3. Якщо є обидві дитини.

Розглянемо окремо перші два випадки видалення вершини `x`.

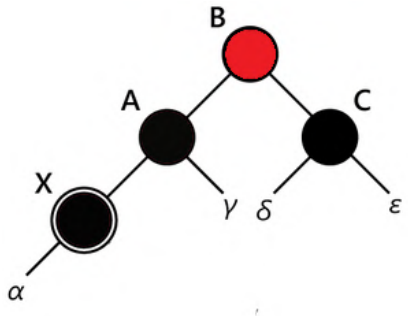
1. Якщо вершина `x` не має дітей,



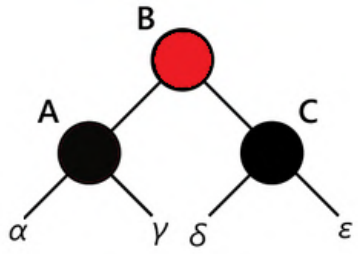
то змінюємо вказівник на неї у батька на nul.



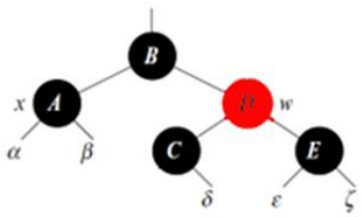
2. Якщо у вершини **x** лише одна дитина,



то робимо у її батька посилання на дитину вершини **x** замість цієї вершини **x**.



3. Якщо є обидві дитини, то знаходимо вершину з наступним значенням ключа (на рисунку верщина **c**.



У такої вершини немає лівої дитини, оскільки така вершина знаходиться в правому піддереві вихідної вершини і вона найлівіша в ньому.

Ми копіюємо дані з `c` в `x`, і тепер наше завдання зводиться до видалення вузла `c`, який гарантовано має не більше одного нащадка. Ці випадки ми вже розглянули.

1.3. Коли виникає проблема?

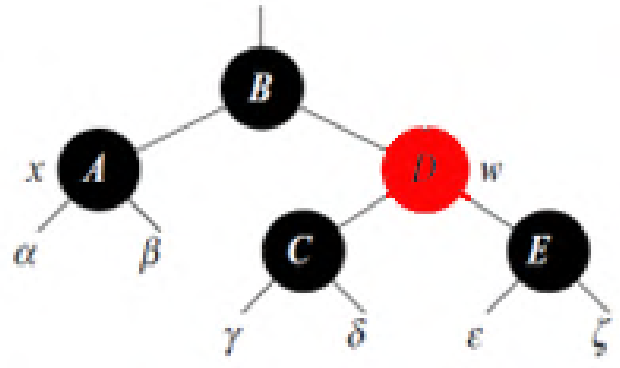
- Якщо видалений вузол `c` був червоним: Червоні вузли не впливають на чорну висоту. Ми просто замінюємо `y` на його нащадка `x` (який обов'язково є чорним листком `NULL`), і всі правила залишаються в силі.
- Якщо видалений вузол `y` був чорним: Це і є джерело всіх проблем. Видалення чорного вузла порушує правило чорної висоти. Щоб це виправити, ми уявляємо, що вузол `x`, який зайняв місце `y`, тепер має "подвійний чорний" колір. Якщо `x` був червоним, він просто стає чорним, і проблема вирішена. Але якщо `x` теж був чорним (або `NULL`), ми маємо "подвійно чорний" вузол, і цю "зайву чорноту" потрібно усунути.

1.4. Алгоритм відновлення: позбуваємось "подвійного чорного"

Під «відновленням» ми будемо розуміти процес перефарбування або переносу догори чорного кольору для того, щоб після видалення чорного вузла відновити властивості червоно-чорного дерева.

Позначимо наш проблемний вузол через `x` . Ми будемо виконувати цикл відновлення, доки наш проблемний вузол `x` є "подвійно чорним" і не є коренем. На кожній ітерації ми розглядаємо брата (sibling) нашого вузла `x`, якого позначимо як `w`.

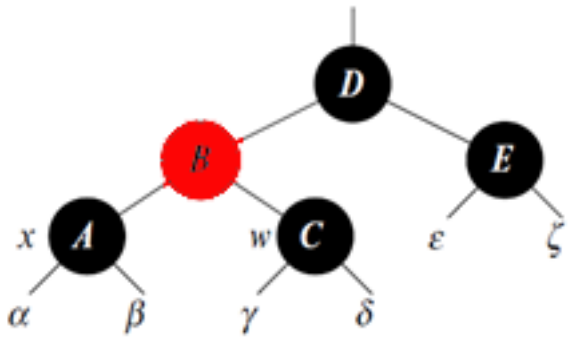
Нехай x вказує на подвійний чорний колір, w вказує на брата x. Незафарбований вузол має червоний або чорний колір.



Випадок 1: вузол w червоний.

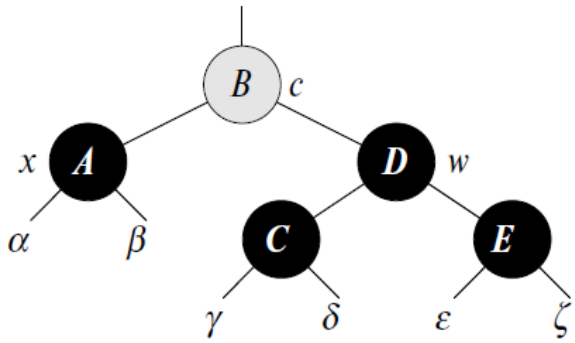
Випадок 1 виникає, коли вузол w (брат вузла x) - червоний. Оскільки w повинен мати чорних нащадків, можна обміняти кольори w та

батьківського x, а потім виконати лівий поворот навколо батьківського x без порушення будь-яких червоно-чорних властивостей.



Новий “брат” x – тепер чорний. Таким шляхом випадок 1 наводиться до випадку 2, 3 або 4. І потрібні подальші дії видалення подвійного чорного.

Випадок 2: вузол w чорний, обидва його дочірні вузли чорні.

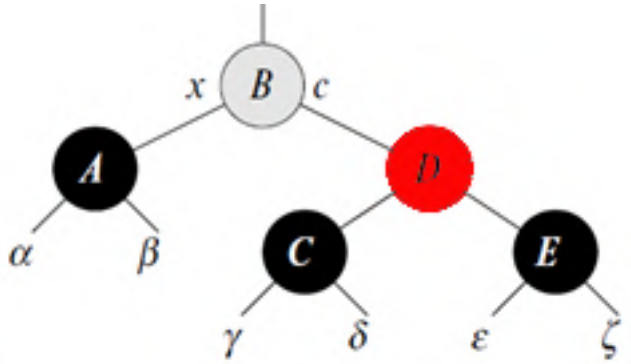


Оскільки вузол w чорний, ми забираємо одне чорне забарвлення у x, а w робимо червоним.

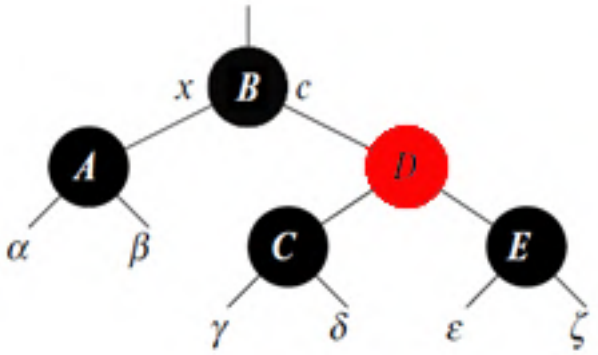
Нагадаємо, що подвійний чорний колір ми запровадили для збереження чорної висоти дерева.

Щоб компенсувати видалення чорного забарвлення у x і w, ми можемо додати додатковий чорний колір батьківському вузлу x, який до цього міг бути як червоним - якщо ми перейшли до Випадку 2 від Випадку 1, так і чорним – якщо ми розглядаємо Випадок 2 як самостійний.

І переносимо покажчик x на батьківський вузол:

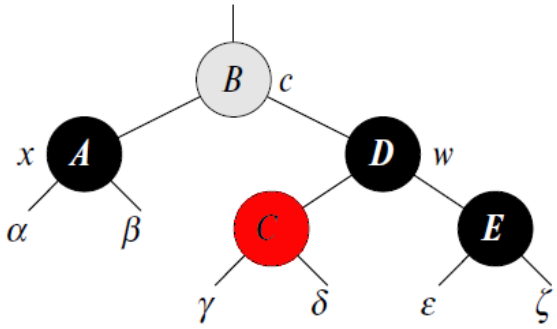


Якщо ми перейшли до Випадку 2 від Випадку 1, то забарвлюємо вузол x в чорний колір.

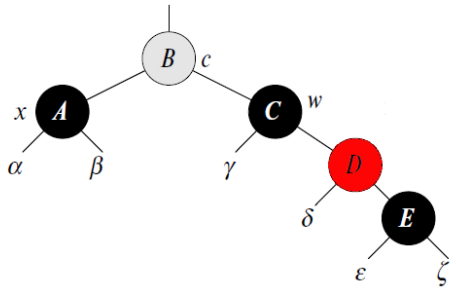


Якщо ми розглядаємо Випадок 2 як самостійний, то вузол x набуває подвійного чорного кольору.

Випадок 3: вузол w чорний, його лівий дочірній вузол червоний, а правий - чорний.

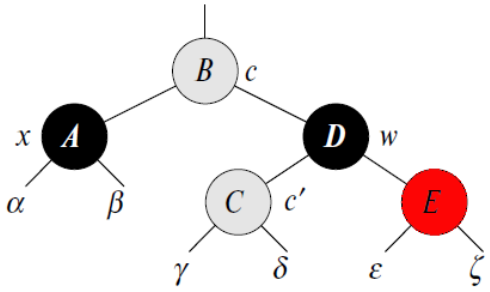


Ми змінюємо кольори w та лівого дочірнього w, а потім виконати правий поворот навколо w без порушення будь-яких червоно-чорних властивостей.

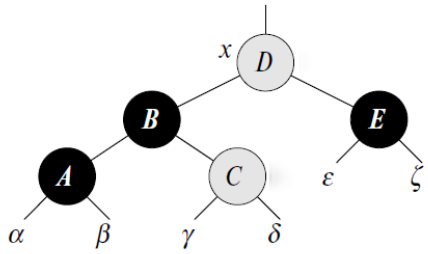


Новим “братом” вузла x після цього буде чорний вузол з червоним правим дочірнім вузлом, таким чином ми переходимо від випадку 3 до 4 випадку.

Випадок 4: вузол w чорний, його правий дочірній вузол червоний



Виконуємо обмін квітів у батьківського вузла x та у брата x. Потім робимо лівий поворот довкола батьківського вузла x. Тепер ми можемо усунути зайву чорноту в x, роблячи його просто чорним, без порушення будь-яких червоно-чорних властивостей. Переносимо вказівник x нагору. Якщо x вказує на корінь дерева, то фарбуємо його в чорний колір і приходимо до завершення роботи.



Якщо ні, то перевіряємо властивості червоного чорного дерева далі.

При цих перетвореннях у кожному випадку кількість чорних вузлів (включаючи додаткову чорноту x) на шляху від кореня включно до кожного з піддерев α, β, . . . , залишається незмінним. Тим самим зберігається властивість 5 червоно-чорного дерева.

Наведемо псевдокод основної логіки:

```
ФУНКЦІЯ Delete(key):
    z = знайти_вузол_з_ключем(key)
```

```
    // ... стандартна логіка BST для визначення y (фізично видаляєтьс
я) та x (його нащадок) ...
    y_original_color = y.color

    // ... логіка "пересадки" вузла x на місце y ...

    ВИДАЛИТИ y

    // Якщо видалили чорний вузол, запускаємо відновлення
    ЯКЩО y_original_color == ЧОРНИЙ ТО
        DeleteFixup(x)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ DeleteFixup(x):
    // Цикл триває, доки 'x' не стане коренем і все ще має "зайвий чо
рний" колір
    ПОКИ x != root І x.color == ЧОРНИЙ:
        // Розглядаємо два великі симетричні випадки: 'x' є лівим або
правим нащадком.
        // Розглянемо випадок, коли x є ЛІВИМ нащадком.
        ЯКЩО x == x.parent.left ТО
            w = x.parent.right // Брат 'x'

            // --- Випадок 1: Брат 'w' червоний ---
            ЯКЩО w.color == ЧЕРВОНИЙ ТО
                w.color = ЧОРНИЙ
                x.parent.color = ЧЕРВОНИЙ
                leftRotate(x.parent)
                w = x.parent.right // Оновлюємо вказівник на нового б
рата
            КІНЕЦЬ ЯКЩО

            // --- Випадок 2: Брат 'w' чорний, і обидва його нащадки
чорні ---
            ЯКЩО w.left.color == ЧОРНИЙ І w.right.color == ЧОРНИЙ ТО
                w.color = ЧЕРВОНИЙ
                x = x.parent // Переміщуємо проблему вгору
            // --- Випадки 3 та 4: Брат 'w' чорний, і хоча б один йог
о нащадок червоний ---
            ІНАКШЕ:
                // Випадок 3: Лівий нащадок 'w' червоний (трикутник)
                ЯКЩО w.right.color == ЧОРНИЙ ТО
                    w.left.color = ЧОРНИЙ
                    w.color = ЧЕРВОНИЙ
```

```
                    rightRotate(w)
                    w = x.parent.right // Оновлюємо брата
                КІНЕЦЬ ЯКЩО

                // Випадок 4: Правий нащадок 'w' червоний (лінія)
                w.color = x.parent.color
                x.parent.color = ЧОРНИЙ
                w.right.color = ЧОРНИЙ
                leftRotate(x.parent)
                x = root // Проблема вирішена, виходимо з циклу
            КІНЕЦЬ ІНАКШЕ
        ІНАКШЕ // 'x' є ПРАВИМ нащадком (дзеркальна логіка)
            // ... аналогічні 4 випадки з rightRotate та leftRotate ...
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ

    // Якщо проблема "дійшла" до кореня, він просто стає чорним
    x.color = ЧОРНИЙ
КІНЕЦЬ ФУНКЦІЇ
```

Обходи дерева (Traversals)

Алгоритми обходу (In-order, Pre-order, Post-order, Level-order) для червоно-чорного дерева є абсолютно ідентичними до алгоритмів для звичайного бінарного дерева.

Складність: $O(n)$

2. Область застосування

Червоно-чорні дерева — це “робоча конячка” у світі збалансованих дерев. Вони пропонують чудовий баланс між швидкістю пошуку та вартістю модифікацій.

- Реалізація стандартних бібліотек: Це основний вибір для реалізації впорядкованих асоціативних контейнерів.
- Ядра операційних систем: Використовуються для управління ресурсами, наприклад, для планування процесів.
- Функціональне програмування: Основа для реалізації персистентних структур даних.
- Бази даних: У деяких випадках можуть використовуватися для індексації.

3. Приклади застосування

– C++ STL: `std::map`, `std::set`, `std::multimap`, `std::multiset` є реалізаціями червоно-чорних дерев.
– Java: `java.util.TreeMap`, `java.util.TreeSet`.
– Ядро Linux: “Completely Fair Scheduler” (CFS) використовує червоно-чорне дерево для ефективного вибору процесу з найменшим часом виконання для наступного запуску.

Переваги та недоліки

Переваги: Головна перевага червоно-чорного дерева — це його прагматичний баланс. Воно гарантує логарифмічну складність $O(\log n)$ для всіх основних операцій у найгіршому випадку. На відміну від AVL-дерева, воно вимагає менше операцій обертання при вставках та видаленнях (максимум 2 для вставки, 3 для видалення), хоча і може виконувати більше перефарбовувань. Це робить його швидшим на практиці для завдань з великою кількістю модифікацій.

Недоліки: Червоно-чорне дерево є менш строго збалансованим, ніж AVL-дерево. Його висота може бути до $2\log_2(n+1)$, що трохи більше, ніж у AVL. Це означає, що для завдань, де операції пошуку значно переважають над операціями запису, AVL-дерево може бути трохи швидшим. Реалізація, особливо операції видалення, є дуже складною і вимагає уважної обробки багатьох випадків.

Контрольні запитання

1. Назвіть 5 правил червоно-чорного дерева.
2. Поясніть, що таке “чорна висота” і яка її роль у балансуванні.
3. Чому новий вузол при вставці завжди зафарбовується у червоний колір?
4. Які два основні інструменти використовуються для відновлення балансу в RBT?
5. Порівняйте червоно-чорне дерево та AVL-дерево. У якому випадку RBT буде кращим вибором?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація червоно-чорного дерева та операції вставки.

2. Мета

Практичне засвоєння логіки самобалансування на основі правил кольорів та обертань. Розуміння відмінностей у підходах до балансування порівняно з AVL-деревом.

3. Завдання

Створити програму, що реалізує червоно-чорне дерево та операцію вставки з подальшим відновленням балансу. Операція видалення є необов'язковою через її високу складність.

4. Опис

Програма має складатися з класу/структури `RBTNode` (з полями `key`, `color`, `left`, `right`, `parent`) та основного класу `RedBlackTree`.

5. Порядок виконання

1. Створіть клас `RBTNode`.
2. Реалізуйте обертання: Напишіть функції `leftRotate` та `rightRotate`. Вони схожі на версії для AVL, але не оновлюють висоти.
3. Реалізуйте `insert`: Спершу реалізуйте стандартну вставку як у BST. Новий вузол завжди червоний.
4. Реалізуйте `fix_insertion`: Це найскладніша частина. Керуючись псевдокодом, уважно реалізуйте логіку для всіх випадків (залежно від кольору "дядька").
5. Реалізуйте обхід: Додайте метод для симетричного обходу, щоб перевіряти коректність роботи.
6. Протестуйте: Створіть у `main` меню. Перевірте роботу, вставляючи послідовності, які викликають різні сценарії балансування.

6. Результат

Додаток, що реалізує червоно-чорне дерево та коректно підтримує його інваріанти при вставці елементів.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми.
2. Продемонструвати роботу додатку, показавши, як дерево балансується та перефарбовується після різних послідовностей вставок.
3. Бути готовим відповісти на питання:

- Пояснити, як ваша програма обробляє випадок, коли “дядько” нового вузла є червоним.
- Пояснити, як працює будь-яке з обертань у вашому коді.
- Яка максимальна висота може бути у вашого дерева з N вузлами?

Тести

1. Корінь червоно-чорного дерева завжди має бути:

a) Червоним

b) Чорним

c) Може бути будь-якого кольору

d) Листком

2. Яке з тверджень про червоно-чорні дерева є хибним?

a) Не може бути двох червоних вузлів поспіль.

b) Всі шляхи від вузла до його листків містять однакову кількість чорних вузлів.

c) Новий вузол при вставці завжди зафарбовується у чорний колір.

d) Листки (NIL) вважаються чорними.

3. Яка гарантована часова складність операції пошуку в червоно-чорному дереві з N вузлами?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(N \log N)$

4. У порівнянні з AVL-деревом, червоно-чорне дерево зазвичай:

a) Має меншу висоту.

b) Виконує менше обертань при вставці.

c) Простіше в реалізації.

d) Не підтримує видалення.

5. Якщо у червоно-чорному дереві батько вузла є червоним, то його “дідусь”:

a) Гарантовано червоний.

b) Гарантовано чорний.

c) Може бути будь-якого кольору.

d) Не існує.

Відповіді на прості тести: 1-b, 2-c, 3-b, 4-b, 5-b (бо корінь завжди чорний, а два червоні вузли поспіль неможливі).

Тести підвищеної складності

1. Яка максимальна можлива висота червоно-чорного дерева з N вузлами?

a) $N - 1$

b) $1.44\log_2(N + 1)$

c) $2\log_2(N + 1)$

d) $\log_2 N$

2. Після операції вставки в червоно-чорне дерево, в якому випадку для відновлення балансу достатньо лише перефарбовувань, без обертань?

ss) Коли “дядько” нового вузла чорний, а вузол утворює “лінію”.

tt) Коли “дядько” нового вузла чорний, а вузол утворює “трикутник”.

uu) Коли “дядько” нового вузла червоний.

vv) Обертання потрібні завжди.

3. Яка амортизаційна складність операції вставки в червоно-чорне дерево?

a) $O(\log N)$

b) $O(1)$

c) $O(\log\log N)$

d) $O(N)$

4. В якому випадку червоно-чорне дерево буде повністю ідентичним за структурою та кольорами до повного бінарного дерева?

a) Коли всі вузли є чорними.

b) Коли кількість чорних вузлів на будь-якому шляху дорівнює висоті.

c) Це можливо, якщо побудувати дерево з $2^k - 1$ вузлів і зафарбувати всі вузли в чорний колір.

d) Таке неможливо.

5. “Ліво-орієнтоване червоно-чорне дерево” (LLRB) — це варіація RBT. Яку додаткову умову воно накладає?

a) Всі червоні вузли можуть бути лише правими нащадками.

b) Всі червоні вузли можуть бути лише лівими нащадками.

c) Всі листки мають бути червоними.

d) Дерево не може мати червоних вузлів.

6. Якщо операція `search` в RBT має таку ж складність $O(\log N)$, як і в AVL-дереві, чому на практиці пошук в AVL-дереві зазвичай трохи швидший?

a) Тому що AVL-дерево має простішу реалізацію.

b) Тому що висота AVL-дерева в середньому є меншою, ніж у RBT, через більш строгі вимоги до балансу.

c) Тому що в RBT потрібно додатково перевіряти колір вузла.

d) Це неправда, пошук в RBT завжди швидший.

7. Видалення якого вузла з червоно-чорного дерева є “найпростішим” з точки зору балансування?

a) Чорного вузла, що має двох чорних нащадків.

b) Чорного вузла, що має одного червоного нащадка.

c) Червоного вузла, що є листком.

d) Кореневого вузла.

8. Який з АТД, реалізованих стандартною бібліотекою C++, найімовірніше використовує червоно-чорне дерево “під капотом”?

a) `std::vector`

b) `std::unordered_map`

c) `std::map`

d) `std::priority_queue`

Відповіді.

1. *c)* Коротке пояснення: Це фундаментальна властивість RBT. Через те, що на будь-якому шляху не може бути двох червоних вузлів поспіль, найдовший шлях (де чергуються чорні та червоні) не може бути довшим, ніж удвічі за найкоротший (лише чорні вузли). Це обмежує висоту.

2. *c)* Якщо “дядько” червоний, ми просто перефарбовуємо батька і дядька в чорний, а дідуся — в червоний. Це відновлює правило про відсутність двох червоних поспіль на поточному рівні, “проштовхуючи” проблему на два рівні вгору.

3. *b)* Коротке пояснення: Хоча пошук місця для вставки займає $O(\log N)$, сама кількість операцій балансування (обертань та перефарбовувань) в середньому є амортизованою константою, $O(1)$. Це одна з ключових переваг RBT.

4. *c)* Якщо побудувати ідеально збалансоване дерево, де всі шляхи мають однакову довжину, і зафарбувати всі вузли в чорний, всі 5 правил RBT будуть виконані.

5. *b)* LLRB — це спрощена версія RBT, яка забороняє існування червоних правих нащадків. Це значно зменшує кількість випадків, які потрібно обробляти при вставці та видаленні, спрощуючи реалізацію.

6. *b)* Хоча обидва дерева мають логарифмічну висоту, строгий інваріант AVL-дерева ($h_{left} - h_{right} \in -1,0,1$) робить його більш “компактним”. Менша висота означає меншу максимальну кількість порівнянь при пошуку.

7. *c)* Видалення червоного листка не порушує жодного з правил червоно-чорного дерева. Кількість чорних вузлів на всіх шляхах залишається незмінною, і не може виникнути ситуації “два червоні поспіль”. Тому жодних операцій балансування не потрібно.

8. *c)* `std::map` — це впорядкований асоціативний контейнер, що вимагає логарифмічної складності для операцій. Червоно-чорне дерево ідеально підходить для цих вимог. `std::unordered_map` використовує хеш-таблицю.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте клас червоно-чорного дерева з операцією вставки та всіма сценаріями балансування.
2. Ручне трасування. Вручну намалюйте, як буде змінюватися RBT при послідовній вставці ключів: 10, 85, 15, 70, 20, 60, 30, 50. Покажіть усі обертання та перефарбовування.
3. Реалізація видалення. *(Завдання підвищеної складності)*. Додайте до вашого класу метод `delete`, що підтримує балансування після видалення.

4. Підрахунок обертань. Модифікуйте вашу реалізацію так, щоб вона підраховувала загальну кількість обертань. Порівняйте її з кількістю обертань в AVL-дереві для однакових послідовностей вставок.
5. Чорна висота. Напишіть функцію, що перевіряє, чи є задане червоно-чорне дерево коректним, перевіряючи всі 5 правил, включаючи обчислення чорної висоти.
6. Ітеративна вставка. Спробуйте реалізувати ітеративну (нерекурсивну) версію методу вставки.
7. Порівняння з AVL. Проведіть емпіричний аналіз: порівняйте час побудови (вставки N елементів) для RBT та AVL-дерева на випадкових та відсортованих даних.
8. Перетворення на LLRB. Модифікуйте вашу реалізацію RBT так, щоб вона відповідала правилам ліво-орієнтованого червоно-чорного дерева.
9. Есе-аналіз. Напишіть есе на тему “Чому червоно-чорні дерева стали промисловим стандартом, витіснивши AVL-дерева у багатьох застосуваннях?”.
10. Пошук k-ої статистики. Дослідіть, як можна модифікувати вузол RBT, щоб за час $O(\log n)$ знаходити k-тий за величиною елемент у дереві.

Splay-дерево (Розширюване дерево)

Ми продовжуємо досліджувати світ збалансованих дерев. Розглянувши AVL-дерева, що підтримують баланс за строгими правилами висоти, та червоно-чорні дерева, що використовують гнучкі правила кольорів, ми переходимо до найбільш незвичайного та динамічного представника цієї родини.

Splay-дерево відмовляється від будь-яких явних полів для балансування і натомість використовує “опортуністичну” стратегію, яка оптимізує дерево на основі реальних патернів доступу.

1. Теоретичні відомості

1.1. Ідея Splay-дерева: самооптимізація на льоту

Усі збалансовані дерева, які ми розглядали досі, підтримували свою структуру, використовуючи додаткову інформацію, збережену у вузлах —

висоту для AVL-дерев, колір для червоно-чорних. Splay-дерево йде зовсім іншим шляхом. Воно не зберігає жодних додаткових атрибутів для балансування. Його магія полягає в одній простій, але потужній операції — `splay` -розширення.

Основна ідея: Щоразу, коли ми виконуємо будь-яку операцію над вузлом (пошук, вставка, видалення), ми за допомогою серії обертань переміщуємо цей вузол у корінь дерева.

Ця стратегія базується на емпіричному спостереженні, відомому як принцип локальності посилань: якщо ми щойно звернулися до якогось елемента, велика ймовірність, що ми звернемося до нього або до його "сусідів" знову найближчим часом. Переміщуючи елемент, до якого часто звертаються, у корінь, ми робимо наступний доступ до нього надзвичайно швидким (за $O(1)$). По суті, Splay-дерево постійно адаптується до патернів запитів, тримаючи "гарячі", часто використовувані дані, якомога ближче до вершини.

1.2. Операція `splay` та її сценарії

Операція `splay(x)` переміщує вузол `x` у корінь за допомогою послідовності обертань. Важливо, що це не просто серія звичайних обертань, які ми бачили в AVL-деревах. Наївне переміщення вгору може погіршити структуру дерева. Splay-дерево використовує три специфічні типи подвійних обертань.

Розглянемо вузол `x`, який ми хочемо "підняти", його батька `p` та "дідуся" `g`.

- Сценарій 1: Zig
    - Коли застосовується: Коли батько `p` є коренем дерева.
    - Дії: Виконується одне просте обертання (ліве або праве) навколо `p`, щоб зробити `x` новим коренем.
- Сценарій 2: Zig-Zig ("лінія")
    - Коли застосовується: Коли `x` та його батько `p` обидва є лівими (або обидва є правими) нащадками.
    - Дії: Виконується два обертання в одному напрямку. Спочатку виконується обертання навколо "дідуся" `g`, а потім — навколо "батька" `p`. Це ефективно "підтягує" всю гілку вгору.
- Сценарій 3: Zig-Zag ("трикутник")

- Коли застосовується: Коли `x` є правим нащадком, а `p` — лівим (або навпаки).
- Дії: Виконується подвійне обертання. Спочатку виконується обертання навколо `p`, яке перетворює “трикутник” на “лінію”, а потім — обертання навколо `g`.

Розглянемо більш детально реалізацію цих Сценаріїв.

Сценарій 1: Zig

Сценарій Zig — це фінальний крок у будь-якій операції `splay`. Він застосовується тоді, коли вузол `x`, який ми “піднімаємо”, вже знаходиться дуже близько до вершини дерева, а саме, коли його батько `p` є коренем всього дерева.

У цій ситуації `x` не має “дідуся”, тому складніші комбінації обертань, як-от Zig-Zig чи Zig-Zag, неможливі. Все, що нам потрібно зробити — це виконати одне-єдине обертання, яке поміняє місцями `x` та його батька `p`, зробивши `x` новим коренем дерева.

Мета цього кроку — завершити операцію `splay`, помістивши цільовий вузол `x` на найвищу позицію, тобто зробивши його коренем дерева.

Тут беруть участь лише два вузли:

– `p` — корінь дерева (і батько `x`).

– `x` — нащадок `p`, якого ми піднімаємо.

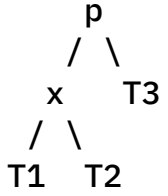
Існує дві дзеркальні конфігурації:

1. Лівий Zig: Вузол `x` є лівим нащадком кореня `p`. Для виправлення потрібне одне праве обертання навколо `p`.
2. Правий Zig: Вузол `x` є правим нащадком кореня `p`. Для виправлення потрібне одне ліве обертання навколо `p`.

Розглянемо покроково процедуру та псевдокод.

Розглянемо випадок Лівого Zig.

Початкова конфігурація така: `p` є коренем, `x` — його лівий нащадок. `T1`, `T2`, `T3` — це відповідні піддерева.



Дії: Ми виконуємо одне праве обертання навколо вузла **p**.

1. Праве піддерево вузла **x** (тобто **T2**) стає новим лівим нащадком вузла **p**. Це коректно, оскільки всі елементи в **T2** більші за **x**, але менші за **p**.
2. Вузол **p** стає правим нащадком вузла **x**.
3. Вузол **x** стає новим коренем всього дерева.

Фінальна конфігурація:

```
      x
     / \
    T1  p
       / \
      T2  T3
```

Властивість бінарного дерева пошуку збережена (**T1 < x < T2 < p < T3**), а вузол **x** тепер знаходиться в корені.

Псевдокод для Лівого Zig (праве обертання):

```
// Вхід: 'p' - корінь дерева, який є батьком вузла, що піднімається.
ФУНКЦІЯ rightRotate(p):
    // Ідентифікуємо 'x' та його праве піддерево 'T2'.
    x = p.left
    T2 = x.right

    // "Усиновлення" T2: T2 стає лівим нащадком p.
    p.left = T2
    ЯКЩО T2 != NULL ТО
        T2.parent = p
    КІНЕЦЬ ЯКЩО

    // Оновлюємо батьківський зв'язок для x. Оскільки p був коренем,
    // новий батько x - NULL, а x стає новим коренем.
    x.parent = p.parent // p.parent тут NULL

    // x стає новим коренем
    root = x

    // 'p' стає правим нащадком 'x'.
    x.right = p
    p.parent = x

    ПОВЕРНУТИ x // Повертаємо новий корінь
КІНЕЦЬ ФУНКЦІЇ
```

Процедура для Правого Zig є дзеркально симетричною і вимагає одного лівого обертання навколо p.

Розглянувши найпростіший випадок Zig, перейдемо до більш складних та поширених сценаріїв, які є серцем механізму самооптимізації Splay-дерева.

Сценарій 2: Zig-Zig ("лінія")

Сценарій Zig-Zig застосовується тоді, коли вузол, який ми "піднімаємо" (x), його батько (p) та "дідусь" (g) утворюють пряму лінію.

Це означає дві можливі конфігурації:

1. Ліво-Ліва (Left-Left): x є лівим нащадком p, а p — лівим нащадком g.
2. Право-Права (Right-Right): x є правим нащадком p, а p — правим нащадком g.

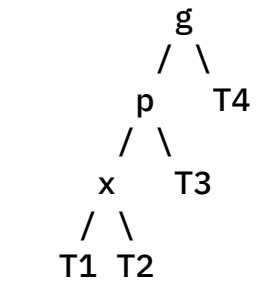
Наївним підходом було б просто виконати два послідовні Zig-обертання. Однак це може не покращити, а навіть погіршити баланс дерева. Стратегія Zig-Zig використовує іншу послідовність обертань, яка значно ефективніше "підтягує" дерево, роблячи його більш збалансованим.

Головна мета — перемістити вузол x на два рівні вгору, на місце його "дідуся" g, при цьому покращивши баланс шляху, на якому знаходились ці три вузли.

Розглянемо випадок Ліво-Лівої (Left-Left) конфігурації. У ньому беруть участь:

- g — "дідусь", найвищий вузол у трійці.
- p — "батько", лівий нащадок g.
- x — цільовий вузол, лівий нащадок p.
- T1, T2, T3, T4 — чотири піддерева, приєднані до цих вузлів.

Початкова конфігурація така:



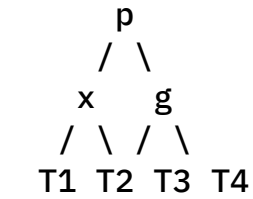
Розглянемо покроково процедуру обертання.

Ключова відмінність від наївного підходу полягає в порядку обертань.

Крок 1: Праве обертання навколо “дідуся” g Спочатку ми виконуємо праве обертання для найвищого вузла g.

- Вузол p піднімається на місце g.
- Вузол g стає правим нащадком p.
- Праве піддерево вузла p (тобто T3) “усиновлюється” вузлом g і стає його новим лівим нащадком.

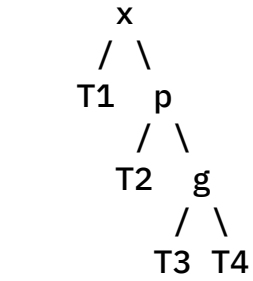
Проміжна конфігурація після Кроку 1:



Крок 2: Праве обертання навколо “нового батька” p Тепер ми виконуємо ще одне праве обертання, але вже навколо вузла p, який тепер є батьком для x.

- Вузол x піднімається на місце p і стає новим коренем цього фрагмента.
- Вузол p стає правим нащадком x.
- Праве піддерево вузла x (тобто T2) стає новим лівим нащадком вузла p.

Фінальна конфігурація така:



Результат: Вузол x успішно перемістився на два рівні вгору. Зверніть увагу, як змінилася структура: дерево стало значно більш збалансованим. Довгий “лівий” шлях перетворився на більш розгалужену та низьку структуру. Властивість бінарного дерева пошуку повністю збережена.

Випадок Право-Правої (Right-Right) конфігурації є дзеркально симетричним і вимагає виконання двох послідовних лівих обертань у тому ж порядку (спочатку навколо g, потім навколо p).

Псевдокод

Хоча операція Zig-Zig складається з двох простих обертань, її логіка в загальній функції `splay` виглядає так:

```
// Фрагмент функції splay(x)
p = x.parent
g = p.parent

// Ліво-Лівий випадок
ЯКЩО p == g.left І x == p.left ТО
    rightRotate(g)
    rightRotate(p)
// Право-Правий випадок
ІНАКШЕ ЯКЩО p == g.right І x == p.right ТО
    leftRotate(g)
    leftRotate(p)
//... інші випадки (Zig-Zag)
КІНЕЦЬ ЯКЩО
```

Ми завершуємо розгляд трьох основних сценаріїв операції `splay`, розбираючи останній, але не менш важливий випадок — Zig-Zag. Ця конфігурація вимагає іншої послідовності обертань, ніж "лінійний" Zig-Zig, і є ключовою для ефективної перебудови дерева.

Сценарій 3: Zig-Zag ("трикутник")

Сценарій Zig-Zag застосовується тоді, коли вузол, який ми "піднімаємо" (`x`), його батько (`p`) та "дідусь" (`g`) утворюють зигзаг або трикутник.

Це означає дві дзеркальні конфігурації:

1. Ліво-Права (Left-Right): `x` є правим нащадком `p`, а `p` — лівим нащадком `g`.
2. Право-Ліва (Right-Left): `x` є лівим нащадком `p`, а `p` — правим нащадком `g`.

Цей випадок схожий на подвійні обертання в AVL-деревах. Наївний підхід тут також не спрацює. Правильна послідовність обертань дозволяє ефективно "випрямити" зигзаг і підняти цільовий вузол `x` на два рівні вгору.

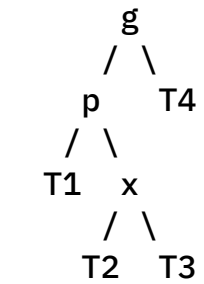
Головна мета — перемістити вузол `x` на місце його "дідуся" `g`, перетворивши незручну зигзагоподібну структуру на більш збалансовану.

Дійові особи та початкова структура

Розглянемо випадок Ліво-Правої (Left-Right) конфігурації. У ньому беруть участь:

- g — “дідусь”.
- p — “батько”, лівий нащадок g.
- x — цільовий вузол, правий нащадок p.
- T1, T2, T3, T4 — чотири піддерева, приєднані до цих вузлів.

Початкова конфігурація така:



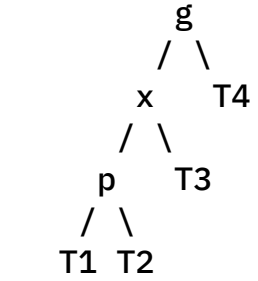
Розглянемо покроково процедуру обертання

Ключовим моментом є те, що обидва обертання виконуються навколо різних вузлів, “випрямляючи” структуру.

Крок 1: Ліве обертання навколо “батька” p Спочатку ми виконуємо ліве обертання для середнього вузла p.

- Вузол x піднімається на місце p.
- Вузол p стає лівим нащадком x.
- Ліве піддерево вузла x (тобто T2) “усиновлюється” вузлом p і стає його новим правим нащадком.

Проміжна конфігурація після Кроку 1: Після цього обертання наша конфігурація “трикутник” перетворилася на знайому нам конфігурацію “лінія” (а саме, Ліво-Ліву).

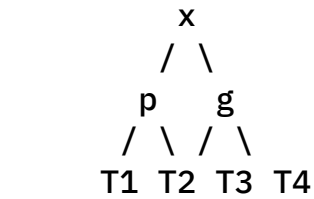


Крок 2: Праве обертання навколо “дідуся” g Тепер, коли ми маємо лінійну структуру, ми виконуємо праве обертання навколо вузла g.

- Вузол x піднімається на місце g і стає новим коренем цього фрагмента.
- Вузол g стає правим нащадком x.

– Праве піддерево вузла `x` (тобто `T3`) стає новим лівим нащадком вузла `g`.

Фінальна конфігурація така:



Результат: Вузол `x` успішно перемістився на два рівні вгору. Зверніть увагу, як `p` та `g` стали його нащадками. Це централізує шлях і значно покращує баланс. Властивість бінарного дерева пошуку повністю збережена.

Випадок Право-Лівої (Right-Left) конфігурації є дзеркально симетричним і вимагає виконання правого обертання навколо `p`, а потім лівого обертання навколо `g`.

Розглянемо псевдокод.

У загальній функції `splay` цей сценарій обробляється таким чином:

```
// Фрагмент функції splay(x)
p = x.parent
g = p.parent

// ... випадки Zig-Zig ...

// Ліво-Правий випадок
ІНАКШЕ ЯКЩО p == g.left І x == p.right ТО
    leftRotate(p)
    rightRotate(g)
// Право-Лівий випадок
ІНАКШЕ ЯКЩО p == g.right І x == p.left ТО
    rightRotate(p)
    leftRotate(g)
КІНЕЦЬ ЯКЩО
```

Таким чином, комбінація цих трьох сценаріїв — Zig, Zig-Zig та Zig-Zag — дозволяє ефективно перемістити будь-який вузол у корінь, одночасно покращуючи загальну структуру дерева.

1.3. Основні операції через призму `splay`

Розберемо, як саме виконуються основні операції в Splay-дереві. Ключова ідея, яку потрібно пам’ятати: кожна операція завершується викликом `splay`, що перебудовує дерево та оптимізує його для майбутніх запитів.

Основні операції та їх реалізація

Розберемо, як саме виконуються основні операції в Splay-дереві. Ключова ідея, яку потрібно пам'ятати: кожна операція завершується викликом `splay`, що перебудовує дерево та оптимізує його для майбутніх запитів.

На відміну від AVL та червоно-чорних дерев, де балансування є реакцією на можливе порушення інваріантів, у Splay-дереві операція перебудови `splay` є невід'ємною частиною кожної основної операції.

Пошук

Пошук у Splay-дереві — це більше, ніж просто знаходження елемента. Це також процес самооптимізації. Коли ви шукаєте елемент, дерево припускає, що ви, ймовірно, захочете звернутися до нього знову, тому воно завбачливо переміщує цей елемент у корінь.

Процедура складається з двох етапів:

1. Стандартний пошук BST: Спочатку ми виконуємо звичайний пошук, як у будь-якому бінарному дереві пошуку. Ми спускаємось по гілках, порівнюючи ключ, доки не знайдемо потрібний вузол. Важливо: якщо ми не знаходимо точного збігу, ми зупиняємось на останньому вузлі, який ми відвідали перед тим, як мали б перейти на `NULL`. Цей вузол є найближчим до нашого ключа.
2. Операція `splay`: Після того, як перший етап завершено, ми беремо останній відвіданий вузол і застосовуємо до нього операцію `splay`. Це перемістить його в корінь дерева за допомогою серії Zig, Zig-Zig та Zig-Zag обертань.

Після завершення `splay` нам залишається лише перевірити, чи є новий корінь тим елементом, який ми шукали.

Псевдокод:

```
// Вхід: 'root' - корінь дерева, 'key' - ключ для пошуку.
// Вихід: новий корінь дерева (після splay) та результат пошуку.
ФУНКЦІЯ Search(root, key):
    // 1. Знаходимо вузол або останній елемент на шляху до нього.
    // Цей крок виконується всередині splay або перед ним.
    // Для простоти, уявимо, що splay сам знаходить потрібний вузол.

    // 2. Виконуємо операцію splay. Вона знайде вузол з ключем 'key'
    // або найближчий до нього і зробить його новим коренем.
    new_root = splay(root, key)
```

```
    // 3. Перевіряємо результат.
    ЯКЩО new_root != NULL І new_root.key == key ТО
        // Елемент знайдено, і він тепер у корені.
        ВИВЕСТИ "Елемент знайдено."
        ПОВЕРНУТИ new_root
    ІНАКШЕ
        // Елемент не знайдено, але дерево оптимізовано.
        ВИВЕСТИ "Елемент не знайдено."
        ПОВЕРНУТИ new_root
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Ось детальний аналіз складності для операції пошуку в Splay-дереві, підготовлений у стилі нашого підручника.

Аналіз складності операції пошуку

Амортизована часова складність операції пошуку в Splay-дереві — $O(\log n)$.

Однак, складність окремої операції в найгіршому випадку — $O(n)$.

Давайте розберемо, що це означає і чому існують дві різні оцінки.

Найгірший випадок: $O(n)$

Splay-дерево, на відміну від AVL або червоно-чорних дерев, не дає жорстких гарантій щодо своєї структури в будь-який конкретний момент часу. Воно може бути тимчасово дуже розбалансованим і навіть виродженим у структуру, схожу на зв'язний список.

Уявіть, що ви шукаєте елемент у такому виродженому дереві. Перший етап операції — це звичайний пошук як у BST, який вимагатиме проходу по всьому ланцюжку з n вузлів. Наступна операція `splay` також буде переміщувати вузол з великої глибини, виконуючи близько $n/2$ обертань. Таким чином, одна окрема операція пошуку в такому найгіршому сценарії матиме лінійну складність $O(n)$.

Амортизований аналіз: $O(\log n)$

Це більш практична та реалістична оцінка, яка є ключовою для розуміння Splay-дерев. Хоча одна операція може бути дуже "дорогою", вона одночасно виконує корисну роботу — значно покращує баланс дерева.

Ідея амортизації: "Дорога" операція, яка щойно виконалася на розбалансованому дереві, "платить наперед" за майбутні операції, роблячи дерево більш збалансованим і наступні доступи до нещодавно використаних елементів дешевшими.

Доведено, що для будь-якої послідовності з M операцій (включаючи пошук) над Splay-деревом, що містить до N елементів, загальний час виконання не перевищить $O(M\log N)$.

Якщо розділити цей загальний час на кількість операцій, ми отримаємо середню, або амортизовану, вартість однієї операції, яка складає:

$$O(M\log N)/M = O(\log N)$$

Що це означає на практиці? Це означає, що ви можете бути впевнені: хоча іноді можуть траплятися "повільні" операції пошуку, середня швидкість роботи з деревом на довгій дистанції буде логарифмічною. Splay-дерево саме себе оптимізує під ваші патерни запитів.

Вставка

Вставка в Splay-дерево також використовує `splay` для оптимізації. Замість того, щоб просто додати новий вузол як листок і потім, можливо, довго його балансувати, Splay-дерево спершу "розчищає" місце біля кореня.

Процедура:

1. Якщо дерево порожнє, новий вузол просто стає коренем.
2. Якщо дерево не порожнє, ми виконуємо операцію `splay` для ключа, який хочемо вставити. Це призведе до однієї з двох ситуацій:
    - Збіг знайдено: Якщо вузол з таким ключем вже існує, він стане коренем. Оскільки Splay-дерева зазвичай не підтримують дублікатів, операція на цьому завершується.
    - Збіг не знайдено: У корінь буде переміщено найближчий до нашого ключа елемент (або менший, або більший).
3. Фінальна перебудова: Тепер, коли корінь `root` є найближчим до нашого нового ключа, ми можемо вставити новий вузол `newNode` як новий корінь.
    - Якщо `key < root.key`: `newNode` стає новим коренем. Його правим нащадком стає старий корінь `root`, а лівим — ліве піддерево старого кореня `root.left`. Зв'язок `root` з його лівим піддеревом розривається.
    - Якщо `key > root.key`: `newNode` стає новим коренем. Його лівим нащадком стає старий корінь `root`, а правим — праве піддерево старого кореня `root.right`. Зв'язок `root` з його правим піддеревом розривається.

Псевдокод такий:

```
// Вхід: 'root' - корінь дерева, 'key' - ключ для вставки.
// Вихід: новий корінь дерева.
ФУНКЦІЯ Insert(root, key):
    // 1. Якщо дерево порожнє.
    ЯКЩО root == NULL ТО
        ПОВЕРНУТИ Створити_Вузол(key)
    КІНЕЦЬ ЯКЩО

    // 2. Виконуємо splay, щоб підняти найближчий вузол у корінь.
    root = splay(root, key)

    // Якщо ключ вже існує, нічого не робимо.
    ЯКЩО root.key == key ТО
        ПОВЕРНУТИ root
    КІНЕЦЬ ЯКЩО

    // 3. Створюємо новий вузол, який стане новим коренем.
    newNode = Створити_Вузол(key)

    ЯКЩО key < root.key ТО
        // Новий вузол менший за старий корінь.
        newNode.right = root
        newNode.left = root.left
        root.left = NULL
        root.parent = newNode // Оновлюємо батька для старого кореня
    ІНАКШЕ // key > root.key
        // Новий вузол більший за старий корінь.
        newNode.left = root
        newNode.right = root.right
        root.right = NULL
        root.parent = newNode // Оновлюємо батька для старого кореня
    КІНЕЦЬ ЯКЩО

    ПОВЕРНУТИ newNode
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності операції вставки

Амортизована часова складність операції вставки в Splay-дерево — $O(\log n)$.

Однак, складність окремої операції в найгіршому випадку — $O(n)$.

Давайте розберемо, що це означає.

Найгірший випадок: $O(n)$

Splay-дерево не дає жорстких гарантій щодо своєї структури в будь-який конкретний момент часу. Воно може бути тимчасово розбалансованим і навіть виродженим у зв'язний список.

Уявіть, що дерево має вигляд довгого ланцюжка. Якщо ми вставляємо новий елемент, першим кроком буде операція `splay`, яка спробує перемістити найближчий до нього вузол у корінь. У найгіршому випадку це вимагатиме проходу по всьому ланцюжку з n вузлів, що призведе до лінійної складності $O(n)$ для цієї однієї операції.

Амортизований аналіз: $O(\log n)$

Це найважливіша характеристика Splay-дерева. Хоча одна операція може бути дуже "дорогою", вона одночасно виконує корисну роботу — значно покращує баланс дерева.

Ідея амортизації: "Дорога" операція, яка щойно виконалася, "платить наперед" за майбутні операції, роблячи дерево більш збалансованим і наступні доступи до нещодавно використаних елементів дешевшими.

Доведено, що для будь-якої послідовності з M операцій (вставка, пошук, видалення) над Splay-деревом, що містить до N елементів, загальний час виконання не перевищить $O(M \log N)$.

Якщо розділити цей загальний час на кількість операцій, ми отримаємо середню, або амортизовану, вартість однієї операції, яка складає:

$$O(M \log N)/M = O(\log N)$$

Що це означає на практиці? Ви можете бути впевнені, що хоча іноді можуть траплятися "повільні" операції, середня швидкість роботи з деревом на довгій дистанції буде логарифмічною. Це робить Splay-дерево дуже ефективним для сценаріїв з нерівномірним доступом до даних.

Видалення

Видалення є найбільш елегантною операцією, що демонструє силу `splay`. Вона також складається з декількох кроків.

Процедура:

1. Пошук та `splay`: Спочатку ми шукаємо елемент, який потрібно видалити. Цей пошук автоматично завершується операцією `splay`, яка переміщує цей елемент у корінь. Якщо елемент не знайдено, `splay` перемістить у корінь останній відвіданий вузол, і ми просто перевіримо, що ключ не збігається, і завершимо операцію.

2. Видалення кореня: Тепер задача звелася до простої — видалити корінь дерева. Після цього у нас залишається два незалежних піддерева: ліве `L` та праве `R`.
3. Об'єднання піддерев: Наша фінальна задача — об'єднати `L` та `R` в одне коректне Splay-дерево. Це робиться дуже елегантно:
    a) Ми беремо ліве піддерево `L`.
    b) У цьому піддереві ми знаходимо найбільший елемент (який, як ми знаємо, не має правого нащадка).
    c) Ми виконуємо операцію `splay` для цього найбільшого елемента всередині піддерева `L`. Це зробить його новим коренем цього піддерева.
    d) Тепер у нового кореня `L` гарантовано немає правого нащадка. Ми просто приєднуємо все праве піддерево `R` як його правого нащадка.

Псевдокод такий:

```
// Вхід: 'root' - корінь дерева, 'key' - ключ для видалення.
// Вихід: новий корінь дерева.
ФУНКЦІЯ Delete(root, key):
    ЯКЩО root == NULL ТО ПОВЕРНУТИ NULL КІНЕЦЬ ЯКЩО

    // 1. Пошук та переміщення вузла в корінь
    root = splay(root, key)

    // Якщо елемент не знайдено, нічого не робимо
    ЯКЩО root.key != key ТО
        ПОВЕРНУТИ root
    КІНЕЦЬ ЯКЩО

    // 2. Тепер вузол, що видаляється, є коренем.
    left_subtree = root.left
    right_subtree = root.right
    ВИДАЛИТИ root

    // 3. Об'єднуємо піддерева
    ЯКЩО left_subtree == NULL ТО
        // Якщо лівого піддерева немає, праве стає новим деревом.
        ПОВЕРНУТИ right_subtree
    КІНЕЦЬ ЯКЩО
```

```
    // 3.c. Робимо найбільший елемент у лівому піддереві його новим к
оренем.
    new_root = splay(left_subtree, key) // `key` гарантовано більший
за всі елементи в `left_subtree`

    // 3.d. Приєднуємо праве піддерево.
    new_root.right = right_subtree
    ЯКЩО right_subtree != NULL ТО
        right_subtree.parent = new_root
    КІНЕЦЬ ЯКЩО

    ПОВЕРНУТИ new_root
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності операції видалення

Амортизована часова складність операції видалення в Splay-дереві — $O(\log n)$.

Однак, складність окремої операції в найгіршому випадку — $O(n)$.

Давайте розберемо, чому існують дві різні оцінки.

Найгірший випадок: $O(n)$

Процедура видалення в Splay-дереві складається з декількох кроків, кожен з яких має свою вартість:

1. Перший `splay`: Пошук елемента, який потрібно видалити, і переміщення його в корінь. У найгіршому випадку, якщо дерево вироджене в зв'язний список, цей крок вимагатиме проходу по всіх n елементах, що дає складність $O(n)$.
2. Видалення кореня: Ця операція сама по собі є $O(1)$.
3. Другий `splay`: Після видалення кореня ми беремо ліве піддерево, знаходимо в ньому максимальний елемент і робимо його новим коренем цього піддерева за допомогою ще однієї операції `splay`. У найгіршому випадку це ліве піддерево може містити майже n елементів, і ця операція також може зайняти $O(n)$ часу.

Оскільки загальна складність визначається найдорожчим кроком, складність однієї окремої операції видалення в найгіршому випадку є лінійною — $O(n)$.

Амортизований аналіз: $O(\log n)$

Це більш практична та реалістична оцінка. Хоча одна операція може бути повільною, вона одночасно значно покращує структуру дерева, що робить наступні операції швидшими.

Ідея амортизації: “Дорога” операція, яка щойно виконалася на розбалансованому дереві, “платить наперед”, роблячи дерево більш збалансованим.

Доведено, що для будь-якої послідовності з M операцій (включаючи видалення) над Splay-деревом, загальний час виконання не перевищить $O(M\log N)$. Якщо розділити цей загальний час на кількість операцій, ми отримаємо середню, або амортизовану, вартість однієї операції:

$$O(M\log N)/M = O(\log N)$$

Що це означає на практиці? Ви можете бути впевнені, що хоча іноді можуть траплятися “повільні” операції видалення, середня швидкість роботи з деревом на довгій дистанції буде логарифмічною. Це робить Splay-дерево ефективним, незважаючи на відсутність гарантій для кожної окремої операції.

Обходи дерева (Traversals)

Алгоритми обходу (In-order, Pre-order, Post-order, Level-order) для Splay-дерева є абсолютно ідентичними до алгоритмів для звичайного бінарного дерева. *Складність*: $O(n)$

1.4. Аналіз складності: магія амортизації

Окрема операція `splay` у найгіршому випадку може зайняти $O(n)$ часу (якщо дерево вироджене). Це означає, що і будь-яка операція (пошук, вставка, видалення) також може зайняти $O(n)$.

Тоді в чому сенс? Сенс в амортизаційному аналізі. Хоча одна операція може бути дорогою, вона водночас “покращує” структуру дерева, роблячи наступні операції дешевшими. Доведено, що будь-яка послідовність з M операцій над Splay-деревом з N елементами виконується за час $O(M\log N)$. Це означає, що амортизована вартість однієї операції становить $O(\log N)$.

Це гарантує, що, хоча іноді можуть траплятися “погані” повільні операції, в середньому за довгий час Splay-дерево буде працювати дуже ефективно.

2. Область застосування

Splay-дерева є ідеальними для систем, де патерн доступу до даних є нерівномірним, тобто існують “гарячі”, часто запитувані елементи.

- Кешування: Для реалізації кешів, де часто використовувані дані мають бути доступні максимально швидко.
- Мережеві маршрутизатори: Для кешування шляхів до популярних напрямків.
- Планувальники пам’яті та збирачі сміття: Для відстеження нещодавно використаних блоків пам’яті.

3. Приклади застосування

- Реалізація кешу LRU (Least Recently Used): Хоча класична реалізація — це хеш-таблиця + двозв’язний список, Splay-дерево може запропонувати простішу альтернативу, автоматично тримаючи нещодавно використані елементи біля кореня.
- Словники в динамічних мовах програмування: Де доступ до певних змінних або функцій може бути значно частішим, ніж до інших.

Переваги та недоліки

Переваги:

- Самооптимізація: Автоматично адаптується до патернів доступу, роблячи часті операції швидшими.
- Хороша амортизована продуктивність: Гарантує середню логарифмічну складність без жорстких правил балансування.
- Простота реалізації: Не вимагає зберігання додаткової інформації (кольору чи висоти), що спрощує структуру вузла.
- Ефективність по пам’яті: Немає додаткових витрат на атрибути балансування.

Недоліки:

- Відсутність гарантій для окремої операції: Найгірший випадок для однієї операції — $O(n)$, що робить Splay-дерева непридатними для систем реального часу, де кожна операція має виконуватися за гарантований час.
- Складніший амортизаційний аналіз: Доведення його ефективності вимагає складних математичних методів.

– Дерево може бути сильно розбалансованим у будь-який конкретний момент часу.

5. Контрольні запитання

3. У чому полягає основна ідея, що лежить в основі Splay-дерева?
4. Назвіть три типи операцій, що використовуються в процесі `splay`. Коли застосовується кожна з них?
5. Поясніть, що таке амортизаційна складність $O(\log N)$ для Splay-дерева. Чи означає це, що кожна операція виконується за логарифмічний час?
6. Для якого типу патернів доступу до даних Splay-дерево буде найбільш ефективним?
7. Порівняйте Splay-дерево та AVL-дерево. У якому випадку Splay-дерево буде кращим вибором, а в якому — AVL?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація Splay-дерева та операції `splay`.

2. Мета

Практичне засвоєння механізму самооптимізації Splay-дерева шляхом реалізації операції `splay` та основних операцій (пошук, вставка).

3. Завдання

Створити програму, що реалізує Splay-дерево та дозволяє виконувати операції пошуку та вставки, демонструючи, як дерево змінює свою структуру після кожної операції.

4. Опис

Програма має складатися з класу/структури `SplayNode` (з полями `key`, `left`, `right`, `parent`) та основного класу `SplayTree`.

5. Порядок виконання

8. Створіть клас `SplayNode`.
9. Реалізуйте обертання: Напишіть базові функції `leftRotate` та `rightRotate`.
10. Реалізуйте `splay`: Це найскладніша частина. Напишіть функцію, яка в циклі виконує Zig, Zig-Zig або Zig-Zag обертання, доки потрібний вузол не стане коренем.

11. Реалізуйте `search(key)`: Реалізуйте функцію, яка знаходить вузол (або останній на шляху) і потім викликає для нього `splay`.

12. Реалізуйте `insert(key)`: Реалізуйте вставку, яка спершу робить `splay`, а потім приєднує новий корінь.

13. Реалізуйте візуалізацію: Напишіть просту функцію, що виводить структуру дерева (напр., через обхід з відступами), щоб бачити зміни.

14. Протестуйте: Створіть дерево, а потім послідовно звертайтеся до різних вузлів, виводячи структуру дерева після кожної операції, щоб побачити `splay` в дії.

6. Результат

Додаток, що реалізує Splay-дерево та наочно демонструє, як воно змінюється після операцій.

7. Після виконання лабораторної роботи студент повинен надати:

15. Вихідний код програми.

16. Продемонструвати роботу додатку, показавши, як вузол, до якого звернулися, стає новим коренем.

17. Бути готовим відповісти на питання:

- Пояснити різницю між Zig-Zig та Zig-Zag випадками у вашому коді.
- Чому для Splay-дерева так важливо мати вказівник на батьківський вузол?
- Яка амортизована складність вашої реалізації?

Тести

18. Основна операція, яка лежить в основі Splay-дерева, це:

ww) Перефарбовування вузлів.

xx) Підрахунок висоти.

yy) Переміщення вузла, до якого звернулися, в корінь.

zz) Видалення вузлів.

5. Яка амортизована часова складність операцій у Splay-дереві?

aaa) $O(1)$

bbb) $O(\log N)$

ccc) $O(N)$

ddd) $O(N\log N)$

7. Яке твердження про Splay-дерево є істинним?

eee) Воно завжди є ідеально збалансованим.

fff) Його висота гарантовано не перевищує $2\log N$.

ggg) Воно оптимізується для доступу до елементів, що використовуються часто.

hhh) Воно не є бінарним деревом пошуку.

5. Splay-дерево є найкращим вибором, коли:

iii) Потрібні гарантії виконання кожної операції за логарифмічний час.

jjj) Патерн доступу до даних є нерівномірним (є “гарячі” елементи).

kkk) Дерево створюється один раз і ніколи не змінюється.

lll) Потрібно часто ітерувати по всіх елементах у відсортованому порядку.

7. Який з випадків `splay` застосовується, коли батько вузла є коренем?

mmm) Zig-Zig

nnn) Zig-Zag

ooo) Zig

ppp) Zag

Відповіді на прості тести: 1-c, 2-b, 3-c, 4-b, 5-c.

Тести підвищеної складності

19. Якщо послідовно звертатися до всіх елементів Splay-дерева у відсортованому порядку, до якої структури воно врешті-решт виродиться?

qqq) До повного бінарного дерева.

rrr) До червоно-чорного дерева.

sss) До сильно нахиленого вліво (або вправо) зв’язного списку.

ttt) Його структура не зміниться.

Відповідь: c)

– Коротке пояснення: Коли ми звертаємось до мінімального елемента, він стає коренем. Потім ми звертаємось до наступного, він стає коренем, а попередній корінь — його лівим нащадком. Цей процес,

повторений для всіх елементів, перетворює дерево на довгий ланцюжок.

6. Чому наївне переміщення вузла вгору за допомогою послідовних простих обертань є поганою ідеєю порівняно зі стратегією Zig-Zig/Zig-Zag?

uuu) Це вимагає більше операцій обертання.

vvv) Це не зберігає властивість бінарного дерева пошуку.

www) Це може не покращити, а навіть погіршити баланс дерева, роблячи найдовший шлях ще довшим.

xxx) Це працює повільніше на апаратному рівні.

Відповідь: c)

– Коротке пояснення: Прості послідовні обертання можуть “розтягувати” дерево, тоді як Zig-Zig та Zig-Zag обертання не лише піднімають вузол, але й “підтягують” решту дерева, скорочуючи загальну довжину шляху до вузлів у піддереві.

8. Для якої з наступних операцій Splay-дерево НЕ забезпечує амортизовану складність $O(\log N)$?

yyy) Вставка

zzz) Пошук

aaaa) Видалення

bbbb) Обхід всього дерева (Traversal)

Відповідь: d)

– Коротке пояснення: Будь-який обхід вимагає відвідування всіх N вузлів, тому його складність за визначенням є $O(N)$, незалежно від структури дерева.

6. Splay-дерево ідеально підходить для реалізації кешу. Який тип кешу воно імітує найкраще?

cccc) FIFO (First-In, First-Out)

dddd) LIFO (Last-In, First-Out)

eeee) LRU (Least Recently Used)

ffff) LFU (Least Frequently Used)

Відповідь: c)

– Коротке пояснення: Переміщуючи елемент, до якого щойно звернулися, в корінь, Splay-дерево автоматично тримає нещодавно використані елементи близько до вершини. Це дуже схоже на поведінку LRU кешу.

8. Яке твердження найбільш точно описує компроміс між Splay-деревом та AVL-деревом?

gggg) Splay-дерево швидше в найгіршому випадку, але повільніше в середньому.

hhhh) AVL-дерево гарантує швидкість кожної окремої операції, тоді як Splay-дерево гарантує лише середню швидкість у довгій послідовності операцій.

iiii) Splay-дерево вимагає більше пам’яті для зберігання кольорів.

jjjj) AVL-дерево краще адаптується до патернів доступу.

Відповідь: b)

– Коротке пояснення: Це ключова відмінність. AVL дає гарантії для кожної операції, що робить його придатним для систем реального часу. Splay-дерево може мати повільні окремі операції, але його загальна продуктивність за довгий час буде високою.

7. Якщо виконати операцію `splay` для вузла, який вже є коренем, що станеться?

kkkk) Дерево буде повністю перебудоване.

llll) Виникне помилка.

mmmm) Нічого не станеться, структура дерева не зміниться.

nnnn) Корінь буде видалено.

Відповідь: c)

– Коротке пояснення: Цикл в операції `splay` триває, доки вузол не стане коренем. Якщо він вже є коренем, цикл не виконається жодного разу, і жодних обертань не відбудеться.

8. Чи можна реалізувати Splay-дерево без вказівників на батьківські вузли?

oooo) Ні, це неможливо.

pppp) Так, але це вимагатиме передачі “стека” батьків у рекурсивну функцію `splay`, що ускладнить реалізацію.

qqqq) Так, це стандартна реалізація.

rrrr) Так, але це погіршить амортизаційну складність до $O(N)$.

Відповідь: b)

– Коротке пояснення: Для виконання обертань потрібен доступ до батька та дідуся. Без прямих вказівників `parent` нам доведеться або зберігати шлях від кореня до вузла, або передавати його через параметри рекурсії, що можливо, але менш елегантно.

9. Яка з операцій у Splay-дереві є “побічним ефектом” усіх інших?

ssss) Пошук

tttt) Вставка

uuuu) Видалення

vvvv) Splay

Відповідь: d)

– Коротке пояснення: Операція `splay` є фундаментальною. Всі інші операції (пошук, вставка, видалення) реалізуються через неї. Вони спершу виконують стандартну логіку BST, а потім викликають `splay` для перебудови дерева.

Завдання для самостійної роботи

20. Реалізація. Повністю реалізуйте клас Splay-дерева з операціями вставки, пошуку та видалення.

21. Ручне трасування. Вручну намалюйте, як буде змінюватися Splay-дерево при послідовній вставці ключів: 10, 20, 30, а потім при пошуку ключа 10.

22. Порівняння з BST. Проведіть експеримент: порівняйте час виконання 1000 послідовних пошуків одного і того ж елемента в Splay-дереві та у статичному збалансованому BST.

23. Аналіз Zig-Zig vs прості обертання. Реалізуйте наївну версію `splay`, яка просто робить послідовні прості обертання. Порівняйте висоту дерева після послідовного доступу до елементів у порівнянні з коректною реалізацією.

24. Операція об’єднання (Join). Дослідіть та реалізуйте операцію `join(T1, T2)`, яка об’єднує два Splay-дерева T1 та T2 (де всі ключі в T1 менші за всі ключі в T2) в одне.

25. Операція розрізання (Split). Дослідіть та реалізуйте операцію `split(T, x)`, яка розрізає дерево T на два дерева: одне з ключами, меншими за x, інше — з більшими.

26. Есе-аналіз. Напишіть есе на тему “Splay-дерево як структура даних для кешування: переваги та недоліки порівняно з LRU-кешем на основі хеш-таблиці та списку”.

27. Доступ по індексу. Чи можна ефективно реалізувати операцію “знайти k-тий за величиною елемент” у Splay-дереві? Обґрунтуйте.

28. Ітеративний Splay. Спробуйте реалізувати ітеративну (нерекурсивну) версію операції `splay`.

29. Персистентність. Чи можна зробити Splay-дерево персистентним? Які складнощі виникнуть при цьому?

Декартове дерево

Розглянувши детерміновані підходи AVL-дерев та червоно-чорних дерев, ми познайомимось з елегантною та часто простішою в реалізації імовірнісною альтернативою. Декартове дерево використовує силу випадковості, щоб досягти балансу, і є чудовим прикладом того, як можна поєднати дві різні структури даних в одну потужну.

1. Теоретичні відомості

1.1. Ідея Декартового дерева: два в одному

Досі ми розглядали дерева, які підтримували баланс за допомогою складних, детермінованих правил: строгий контроль висот в AVL або хитра система кольорів у червоно-чорних деревах. Декартове дерево (Treap), назва якого є поєднанням слів Tree (дерево) та Heap (купа), пропонує принципово інший, імовірнісний підхід.

Основна ідея: Кожен вузол у Декартовому дереві зберігає не лише ключ (key), але і випадково згенерований пріоритет (priority). Структура дерева повинна одночасно задовольняти двом інваріантам:

10. Властивість бінарного дерева пошуку (BST Invariant): Відносно ключів, дерево є звичайним бінарним деревом пошуку. Для будь-якого вузла, всі ключі в його лівому піддереві менші, а в правому — більші.

11. Властивість купи (Heap Invariant): Відносно пріоритетів, дерево є купою (зазвичай max-heap). Для будь-якого вузла, його пріоритет є більшим або рівним за пріоритети обох його нащадків.

Структура вузла Декартового дерева:

```
Структура TreapNode:
    key       // Ключ (дані, що впорядковуються)
    priority  // Випадковий пріоритет
    left      // Вказівник на лівого нащадка
    right     // Вказівник на правого нащадка
    // parent - опціонально, але корисно для деяких реалізацій
```

1.2. Сила випадковості

Магія полягає у випадковості пріоритетів. Коли ми вставляємо новий елемент, ми генеруємо для нього випадковий пріоритет. Цей пріоритет визначає його "вертикальне" положення в дереві.

Можна довести, що структура Декартового дерева з випадковими пріоритетами є такою самою, як структура звичайного BST, побудованого шляхом вставки елементів у порядку, визначеному їхніми пріоритетами. Оскільки пріоритети випадкові, це еквівалентно тому, що ми вставляємо ключі у випадковій перестановці. А як ми знаємо, BST, побудоване на випадкових даних, з дуже високою ймовірністю буде збалансованим.

Таким чином, Декартове дерево — це розумний спосіб змусити будь-яку послідовність вхідних даних поводитися так, ніби вона була випадковою, і тим самим досягти балансу "безкоштовно", без складних правил.

1.3. Фундаментальні операції: Split та Merge

На відміну від AVL та червоно-чорних дерев, які використовують обертання для *реактивного* виправлення балансу, Декартове дерево базується на двох інших, більш композиційних операціях.

Операція `Split` (Розділення)

Операція `Split(T, key)` — це фундаментальний механізм, який лежить в основі гнучкості Декартового дерева. Вона не просто знаходить елемент, а елегантно "розрізає" одне дерево `T` на два нових, базуючись на заданому ключі `key`.

Мета: Отримати два дерева, `L` та `R`, такі що:

– Дерево `L` містить усі вузли з початкового дерева `T`, ключі яких є меншими за `key`.

– Дерево `R` містить усі вузли з початкового дерева `T`, ключі яких є більшими або рівними за `key`.

При цьому обидва нові дерева, `L` та `R`, повинні залишатися коректними Декартовими деревами, тобто зберігати властивості і бінарного дерева пошуку, і купи.

Алгоритм `Split` є рекурсивним і працює, спускаючись від кореня до листків. На кожному кроці він приймає рішення, до якого з двох результуючих дерев (лівого чи правого) належить поточний вузол.

Давайте розглянемо, як функція `Split(t, key)` поводиться для поточного піддерева з коренем `t`.

Базовий випадок: Якщо поточний вузол `t` є порожнім (`NULL`), це означає, що ми дійшли до кінця гілки. Розділяти нічого, тому ми просто повертаємо два порожні дерева (`NULL`, `NULL`).

Рекурсивний крок: Існує два можливих сценарії:

Сценарій 1: Ключ поточного вузла менший за ключ розділення (`t.key < key`)

– Що це означає? Ми точно знаємо, що поточний вузол `t` (разом з усім його лівим піддеревом, де ключі ще менші) має потрапити до лівого результуючого дерева `L`.

– Проблема: А що робити з правим піддеревом вузла `t`? Воно може містити ключі як менші, так і більші за `key`.

– Рішення: Ми маємо рекурсивно розділити праве піддерево `t.right` за тим самим ключем `key`. Цей виклик поверне нам дві частини: `sub_L` (елементи з `t.right`, що менші за `key`) та `sub_R` (елементи з `t.right`, що більші або рівні `key`).

– Збирання результату:

  - Новим правим нащадком вузла `t` стає `sub_L`.
  - Фінальним лівим деревом `L` стає оновлений вузол `t`.
  - Фінальним правим деревом `R` стає `sub_R`.

Сценарій 2: Ключ поточного вузла більший або рівний за ключ розділення (`t.key >= key`)

– Що це означає? Ми точно знаємо, що поточний вузол `t` (разом з усім його правим піддеревом) має потрапити до правого результуючого дерева `R`.

– Проблема: Ліве піддерево вузла `t` може містити елементи, які нам потрібно розділити.
– Рішення: Рекурсивно розділяємо ліве піддерево `t.left` за ключем `key`, отримуючи `sub_L` та `sub_R`.
– Збирання результату:
    - Новим лівим нащадком вузла `t` стає `sub_R`.
    - Фінальним правим деревом `R` стає оновлений вузол `t`.
    - Фінальним лівим деревом `L` стає `sub_L`.

Псевдокод такий

```
// Вхід: t - корінь поточного дерева, key - ключ для розділення.
// Вихід: пара вказівників (left_tree, right_tree).
ФУНКЦІЯ Split(t, key):
    // Базовий випадок: якщо дерево порожнє, повертаємо два порожні д
ерева.
    ЯКЩО t == NULL ТО
        ПОВЕРНУТИ (NULL, NULL)
    КІНЕЦЬ ЯКЩО

    // Сценарій 1: поточний вузол йде в ліве дерево.
    ЯКЩО t.key < key ТО
        // Рекурсивно розділяємо праве піддерево.
        (sub_L, sub_R) = Split(t.right, key)

        // Новим правим нащадком t стає ліва частина його колишнього
правого піддерева.
        t.right = sub_L

        // Повертаємо пару: оновлене ліве дерево (з коренем t) та пра
ве.
        ПОВЕРНУТИ (t, sub_R)

    // Сценарій 2: поточний вузол йде в праве дерево.
    ІНАКШЕ:
        // Рекурсивно розділяємо ліве піддерево.
        (sub_L, sub_R) = Split(t.left, key)

        // Новим лівим нащадком t стає права частина його колишнього
лівого піддерева.
        t.left = sub_R
```

```
        // Повертаємо пару: ліве дерево та оновлене праве (з коренем t).
        ПОВЕРНУТИ (sub_L, t)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності

Алгоритм `Split` на кожному кроці рекурсії спускається на один рівень вглиб дерева. Таким чином, його часова складність напряму залежить від висоти дерева h.

- Очікувана (середня) складність: $O(\log N)$
- Найгірша складність: $O(N)$

Операція `Merge` (Злиття)

Операція `Merge(L, R)` — це елегантний алгоритм, який приймає на вхід два коректних Декартових дерева, `L` та `R`, і об'єднує їх в одне нове, також коректне, Декартове дерево.

Найважливіша вимога (передумова): Ця операція може бути виконана лише за однієї суворої умови: усі ключі в дереві `L` мають бути строго меншими за всі ключі в дереві `R`. Це означає, що ми не можемо злити два довільні дерева, а лише ті, що представляють собою дві послідовні, невпорядковані частини. Саме тому `Merge` є ідеальною парою для операції `Split`, яка якраз і створює такі частини.

Як об'єднати два дерева, не порушивши жодного з двох інваріантів (властивості BST та властивості купи)? Відповідь криється у пріоритетах.

Ідея: Новим коренем об'єднаного дерева має стати той вузол, у якого найбільший пріоритет — або корінь дерева `L`, або корінь дерева `R`. Чому? Тому що це єдиний спосіб задовольнити властивість купи (де батько завжди має вищий пріоритет, ніж його нащадки).

Базові випадки:

- Якщо одне з дерев, наприклад `L`, є порожнім (`NULL`), то результатом злиття очевидно буде друге дерево, `R`.
- Якщо `R` є порожнім, результатом буде `L`.

Рекурсивний крок: Припустимо, обидва дерева не порожні. Ми порівнюємо пріоритети їхніх коренів.

Сценарій 1: Пріоритет кореня `L` вищий (`L.priority > R.priority`)

– Що це означає? Щоб задовольнити властивість купи, корінь дерева `L` повинен стати новим загальним коренем.

– Що ми робимо?

1. Ми фіксуємо корінь `L` як вершину нашого нового дерева. Його ліве піддерево залишається незмінним, оскільки всі ключі там менші за `L.key`, а отже, і за всі ключі в `R`.
2. Тепер нам потрібно приєднати дерево `R`. Оскільки всі ключі в `R` більші за `L.key`, `R` має бути десь у правому піддереві нового кореня.
3. Ми рекурсивно викликаємо `Merge` для початкового правого піддерева `L.right` та всього дерева `R`. Результат цього злиття і стає новим правим нащадком для кореня `L`.

Сценарій 2: Пріоритет кореня `R` вищий або рівний (`R.priority >= L.priority`)

– Що це означає? Дзеркальна ситуація. Корінь дерева `R` стає новим загальним коренем.

– Що ми робимо?

4. Фіксуємо корінь `R` як вершину. Його праве піддерево залишається незмінним.
5. Оскільки всі ключі в `L` менші за `R.key`, `L` має бути десь у лівому піддереві нового кореня.
6. Ми рекурсивно викликаємо `Merge` для всього дерева `L` та початкового лівого піддерева `R.left`. Результат стає новим лівим нащадком для кореня `R`.

Псевдокод

```
// Вхід: l, r - вказівники на корені лівого та правого дерев для злит
тя.
// Передумова: max_key(l) < min_key(r).
// Вихід: вказівник на корінь нового об'єднаного дерева.
ФУНКЦІЯ Merge(l, r):
    // Базові випадки: якщо одне з дерев порожнє, повертаємо інше.
    ЯКЩО l == NULL ТО
        ПОВЕРНУТИ r
    КІНЕЦЬ ЯКЩО
    ЯКЩО r == NULL ТО
        ПОВЕРНУТИ l
```

```
    КІНЕЦЬ ЯКЩО

    // Сценарій 1: корінь лівого дерева стає новим коренем.
    ЯКЩО l.priority > r.priority ТО
        // Його праве піддерево рекурсивно зливається з усім правим д
еревом.
        l.right = Merge(l.right, r)
        ПОВЕРНУТИ l

    // Сценарій 2: корінь правого дерева стає новим коренем.
    ІНАКШЕ:
        // Його ліве піддерево рекурсивно зливається з усім лівим дер
евом.
        r.left = Merge(l, r.left)
        ПОВЕРНУТИ r
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності

Алгоритм `Merge` на кожному кроці рекурсії спускається на один рівень вглиб або по правому контуру лівого дерева, або по лівому контуру правого дерева. Таким чином, його часова складність напряму залежить від висоти дерев, що зливаються.

- Очікувана (середня) складність: $O(\log N)$
- Найгірша складність: $O(N)$

Операція Вставки

На відміну від AVL або червоно-чорних дерев, де новий елемент вставляється як листок з подальшим “ремонтом” дерева шляхом обертань, Декартове дерево використовує більш композиційний підхід. Вставка тут — це витончений процес, що повністю базується на операціях `Split` та `Merge`, які ми вже розглянули.

Щоб вставити новий вузол з ключем `key` у дерево `T`, не порушивши жодного з інваріантів, ми робимо наступне:

1. Ми “розчищаємо” місце для нового вузла. Для цього ми розрізаємо початкове дерево `T` за ключем `key`. В результаті ми отримуємо два абсолютно коректних Декартових дерева: `L` (з усіма ключами, меншими за `key`) та `R` (з усіма ключами, більшими або рівними `key`).

2. Тепер у нас є три компоненти: ліве дерево `L`, праве дерево `R` та наш новий вузол `newNode`. Оскільки `max_key(L) < key < min_key(R)`, ми можемо їх елегантно об'єднати.
3. Ми зливаємо `L` з `newNode` за допомогою операції `Merge`. Оскільки `newNode` має лише один елемент, `Merge(L, newNode)` буде дуже швидким.
4. Потім ми зливаємо результат попереднього кроку з правим деревом `R` за допомогою ще однієї операції `Merge`.

Цей процес гарантує, що властивості і бінарного дерева пошуку, і купи будуть збережені, оскільки операції `Split` та `Merge` самі по собі зберігають ці інваріанти.

Покрокова процедура

1. Створення нового вузла. Створюємо новий вузол `newNode` із заданим ключем `key` та випадково згенерованим пріоритетом.
2. Розділення дерева. Викликаємо функцію `Split(root, key)`, де `root` — корінь нашого поточного дерева. Вона повертає нам два дерева: `left_part` та `right_part`.
3. Перше злиття. Викликаємо `Merge(left_part, newNode)`. Оскільки `newNode` — це новий корінь, він, ймовірно, буде об'єднаний з `left_part` на основі їхніх пріоритетів. Результатом буде нове тимчасове дерево `temp_root`.
4. Друге злиття. Викликаємо `Merge(temp_root, right_part)`. Це об'єднує все в одне фінальне дерево. Результат цієї операції стає новим коренем нашого Декартового дерева.

Псевдокод такий

```
// Вхід: 'root' - корінь дерева, 'key' - ключ для вставки.
// Вихід: новий корінь дерева.
ФУНКЦІЯ Insert(root, key):
    // 1. Створюємо новий вузол з випадковим пріоритетом.
    priority = згенерувати_випадковий_пріоритет()
    newNode = Створити_Вузол(key, priority)

    // 2. Розділяємо існуюче дерево за ключем.
    // L містить все, що < key; R - все, що >= key.
    (L, R) = Split(root, key)

    // 3. Зливаємо ліву частину з новим вузлом.
```

```
    temp_root = Merge(L, newNode)

    // 4. Зливаємо отримане дерево з правою частиною.
    new_root = Merge(temp_root, R)

    ПОВЕРНУТИ new_root
КІНЕЦЬ ФУНКЦІЇ

// Нагадування реалізації Split та Merge:
ФУНКЦІЯ Split(t, key):
    // ... реалізація з попереднього розділу ...
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ Merge(l, r):
    // ... реалізація з попереднього розділу ...
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності

Складність операції вставки повністю визначається складністю операцій `Split` та `Merge`, які вона викликає. Оскільки складність кожної з цих операцій залежить від висоти дерева, ми маємо:

- Очікувана (середня) складність: $O(\log N)$. Оскільки очікувана висота Декартового дерева є логарифмічною, то і середня вартість вставки буде логарифмічною.
- Найгірша складність: $O(N)$. У вкрай рідкісному випадку, коли випадкові пріоритети призводять до виродження дерева, висота стає лінійною, і, відповідно, операція вставки також буде мати лінійну складність.

Операція Видалення

Операція видалення, як і вставка, в Декартовому дереві реалізується надзвичайно елегантно за допомогою вже знайомих нам операцій `Split` та `Merge`. Замість складних маніпуляцій з вузлами та їхніми нащадками, ми просто “вирізаємо” непотрібний нам елемент і “зшиваємо” решту дерева.

Загальна стратегія полягає в тому, щоб ізолювати вузол, який ми хочемо видалити, від решти дерева, а потім об’єднати частини, що залишились.

1. Розрізаємо дерево на три частини:
    - `L`: піддерево з усіма ключами, що менші за ключ видалення `key`.
    - `M`: піддерево, що складається з одного вузла з ключем `key` (якщо він існує).

- `R`: піддерево з усіма ключами, що більші за ключ видалення `key`.

2. Ігноруємо середню частину: Ми просто “забуваємо” про частину `M`.
3. Зливаємо решту: Ми об’єднуємо дерева `L` та `R` за допомогою операції `Merge`. Оскільки всі ключі в `L` гарантовано менші за всі ключі в `R`, ця операція є коректною.

Покрокова процедура

Для реалізації цієї ідеї нам знадобиться два виклики операції `Split`.

Крок 1: Відокремлення “менших” елементів Спочатку ми викликаємо `Split(root, key)`. Ця операція розділить наше дерево на дві частини:

– `left_part`: дерево, що містить всі вузли з ключами, меншими за `key`.

– `right_and_equal_part`: дерево, що містить всі вузли з ключами, більшими або рівними `key`.

Крок 2: Відокремлення “більших” елементів Тепер ми беремо друге дерево (`right_and_equal_part`) і знову розділяємо його, але вже за ключем `key + 1` (або будь-яким значенням, що гарантовано більше за `key`, але менше за наступний можливий ключ). Це дозволяє нам відокремити вузол з ключем `key`:

– `equal_part`: дерево, що складається лише з одного вузла з ключем `key` (якщо він існував).

– `right_part`: дерево, що містить всі вузли з ключами, більшими за `key`.

Крок 3: Фінальне злиття На цьому етапі ми маємо `left_part` та `right_part`. Вузол, який ми хотіли видалити, знаходиться в `equal_part`, який ми просто ігноруємо. Ми викликаємо `Merge(left_part, right_part)`, щоб об’єднати дві крайні частини в одне фінальне, коректне Декартове дерево.

Псевдокод такий

```
// Вхід: 'root' - корінь дерева, 'key' - ключ для видалення.
// Вихід: новий корінь дерева без вузла з ключем 'key'.
ФУНКЦІЯ Delete(root, key):
    // Крок 1: Розділяємо дерево на дві частини: < key та >= key.
    (L, R_and_E) = Split(root, key)

    // Крок 2: Розділяємо праву частину на == key та > key.
    // Використовуємо key + 1, щоб відокремити сам вузол з ключем key
.
```

```
    (E, R) = Split(R_and_E, key + 1)

    // Вузол (або піддерево) E, який ми хотіли видалити, просто ігнор
ується.
    // Ми можемо звільнити пам'ять, яку він займав.
    // ВИДАЛИТИ_ДЕРЕВО(E)

    // Крок 3: Зливаємо ліву та праву частини, що залишились.
    new_root = Merge(L, R)

    ПОВЕРНУТИ new_root
КІНЕЦЬ ФУНКЦІЇ

// Нагадування реалізації Split та Merge:
ФУНКЦІЯ Split(t, key):
    // ... реалізація з попереднього розділу ...
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ Merge(l, r):
    // ... реалізація з попереднього розділу ...
КІНЕЦЬ ФУНКЦІЇ
```

Складність операції видалення повністю визначається складністю операцій **Split** та **Merge**, які вона викликає. Оскільки складність кожної з цих операцій залежить від висоти дерева, ми маємо:

- Очікувана (середня) складність: $O(\log N)$ Ми виконуємо дві операції **Split** та одну **Merge**, кожна з яких має очікувану складність $O(\log N)$. Сумарна складність також буде логарифмічною.
- Найгірша складність: $O(N)$ У вкрай рідкісному випадку, коли дерево вироджене, висота стає лінійною, і, відповідно, кожна з операцій **Split**/**Merge** буде мати лінійну складність.

Пошук та обходи

Ці операції є ідентичними до операцій у звичайному BST.

Аналіз складності операцій

Ефективність Декартового дерева, як і будь-якої деревної структури, напряму залежить від його висоти (h). Однак, на відміну від детермінованих дерев (AVL, RBT), які підтримують низьку висоту за допомогою строгих правил, Декартове дерево покладається на магію ймовірності.

Фундамент аналізу: Очікувана та найгірша висота

– Очікуваний (середній) випадок: Оскільки пріоритети вузлам присвоюються випадково, структура Декартового дерева для будь-якого набору ключів є такою ж, як і структура звичайного BST, побудованого шляхом вставки цих ключів у випадковій перестановці. Математично доведено, що очікувана висота такого дерева є логарифмічною. Таким чином, для Декартового дерева з N вузлами, очікувана висота h становить $O(\log N)$.
– Найгірший випадок: Хоча ймовірність цього вкрай мала, теоретично можливо, що наш генератор випадкових чисел видасть пріоритети у вже відсортованому (або майже відсортованому) порядку. Наприклад, для ключів $k_1 < k_2 < \ldots < k_N$ можуть бути згенеровані пріоритети $p_1 < p_2 < \ldots < p_N$. У цьому випадку, щоб зберегти властивість купи, дерево виродиться у довгий ланцюжок (зв'язний список). Таким чином, висота h у найгіршому випадку становить $O(N)$.

Тепер, маючи ці дві оцінки, проаналізуємо кожну операцію.

Операції `Split` та `Merge`

Ці дві операції є фундаментальними для Декартового дерева. * `Split(T, key)`: Процес розділення завжди рухається по одному шляху від кореня вниз до листка. Кількість кроків дорівнює довжині цього шляху. * `Merge(L, R)`: Процес злиття також рухається вниз по одному з двох шляхів (або по правій гілці дерева L, або по лівій гілці дерева R), порівнюючи пріоритети.

Таким чином, складність обох операцій напряму залежить від висоти дерева. * Очікувана складність: $O(\log N)$. * Найгірша складність: $O(N)$.

Пошук

Ідея аналізу: Алгоритм пошуку в Декартовому дереві є абсолютно ідентичним до алгоритму в звичайному BST і ніяк не залежить від пріоритетів. Ми спускаємось по дереву, порівнюючи ключі. Кількість порівнянь у найгіршому випадку дорівнює висоті дерева.

– Найкращий випадок: $O(1)$ Коли шуканий елемент є коренем дерева.
– Очікуваний (середній) випадок: $O(\log N)$ Оскільки очікувана висота дерева логарифмічна, середня кількість кроків для пошуку також буде логарифмічною.
– Найгірший випадок: $O(N)$ Якщо дерево виродилося у зв'язний список, пошук перетвориться на лінійний перебір.

Вставка

Ідея аналізу: Операція вставки, реалізована через `Split` та `Merge`, складається з трьох кроків:

1. `Split(root, key)`: Розділити дерево на дві частини. Складність — $O(h)$.

2. `Merge(left_part, newNode)`: Злити ліву частину з новим вузлом. Складність — $O(h)$.

3. `Merge(temp_root, right_part)`: Злити результат з правою частиною. Складність — $O(h)$.

Загальна складність є сумою складностей цих операцій:

- Очікувана (середня) складність: $O(\log N) + O(\log N) + O(\log N) = O(\log N)$.
- Найгірша складність: $O(N) + O(N) + O(N) = O(N)$.

Видалення

Ідея аналізу: Операція видалення також базується на `Split` та `Merge`.

1. `Split(root, key)`: Розділити дерево на `<key` та `≥key`. Складність — $O(h)$.

2. `Split(greater_equal_part, key + 1)`: Розділити другу частину, щоб відокремити вузол з ключем `key`. Складність — $O(h)$.

3. `Merge(less_part, greater_part)`: Злити дві крайні частини. Складність — $O(h)$.

– Очікувана (середня) складність: $O(\log N)$.

– Найгірша складність: $O(N)$.

Обходи дерева

Ідея аналізу: Незалежно від типу обходу та форми дерева (збалансованої чи виродженої), ми повинні відвідати кожен вузол рівно один раз.

– Часова складність: Оскільки ми маємо виконати певну константну роботу для кожного з N вузлів, загальна складність будь-якого повного обходу завжди буде $O(N)$.

– Просторова складність:

  - DFS (рекурсивні обходи): Визначається максимальною глибиною стека рекурсії, яка дорівнює висоті дерева. Для очікуваного випадку це $O(\log N)$, для найгіршого — $O(N)$.

- BFS (обхід в ширину): Визначається максимальним розміром черги. У найгіршому випадку (для повного, “кущистого” дерева) в черзі може опинитися близько *N/2* листків. Складність — $O(N)$.

Декартове дерево, незважаючи на свою імовірнісну природу, є надзвичайно ефективною структурою даних. Хоча воно і не дає таких жорстких гарантій для кожної окремої операції, як AVL чи червоно-чорні дерева, його очікувана логарифмічна продуктивність у поєднанні зі значно простішою реалізацією робить його чудовою альтернативою для багатьох практичних задач.

2. Область застосування

- Впорядковані множини та словники: Декартове дерево є чудовою і простішою в реалізації альтернативою червоно-чорним деревам.
- Обчислювальна геометрія.
- Задачі, що вимагають операцій **`Split`** та **`Merge`**: Якщо в задачі потрібно часто розділяти та об’єднувати набори даних, Декартове дерево є ідеальним вибором.

3. Приклади застосування

- Реалізація АТД “Множина” у функціональних мовах програмування.
- Основа для більш складних структур, як-от Декартове дерево по неявному ключу (для моделювання масиву з швидкими операціями вставки/видалення в довільному місці).

Переваги та недоліки

Переваги:

- Простота реалізації: Порівняно з AVL або червоно-чорними деревами, Декартове дерево, особливо його версія на основі **`Split`**/**`Merge`**, є значно простішим для розуміння та написання коду.
- Висока ймовірність хорошої продуктивності: Хоча і немає жорстких гарантій, на практиці дерево майже завжди є добре збалансованим.
- Елегантні операції **`Split`** та **`Merge`**: Дозволяють легко реалізовувати складні операції над множинами.

Недоліки:

- Імовірнісна природа: Продуктивність не є гарантованою для кожної окремої операції, а лише “очікуваною”. Це робить його непридатним для систем жорсткого реального часу.

– Залежність від генератора випадкових чисел: Якість випадкових пріоритетів напряму впливає на баланс дерева.
– Додаткова пам’ять: Потребує зберігання пріоритету для кожного вузла.

Контрольні запитання

1. Назвіть два інваріанти, яким має задовольняти Декартове дерево.
2. Поясніть, чому випадкові пріоритети з високою ймовірністю призводять до збалансованого дерева.
3. Опишіть, як працює операція **`Merge`**. Яка головна умова для її вхідних даних?
4. Як реалізується операція вставки за допомогою **`Split`** та **`Merge`**?
5. Порівняйте Декартове дерево та AVL-дерево. У якому випадку Декартове дерево може бути кращим вибором?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація Декартового дерева з операціями на основі **`Split`** та **`Merge`**.

2. Мета

Практичне засвоєння імовірнісного підходу до балансування дерев. Реалізація ключових операцій **`Split`** та **`Merge`** та побудова на їх основі більш складних операцій.

3. Завдання

Створити програму, що реалізує Декартове дерево для зберігання цілих чисел та надає меню для виконання операцій вставки, видалення та пошуку.

4. Опис

Програма має складатися з класу/структури **`TreapNode`** (з полями **`key`**, **`priority`**, **`left`**, **`right`**) та основного класу **`Treap`**, що інкапсулює логіку. Пріоритети мають генеруватися випадковим чином.

5. Порядок виконання

1. Створіть клас **`TreapNode`**.
2. Реалізуйте **`Split`**: Напишіть рекурсивну функцію **`Split(node, key)`**.

3. Реалізуйте `Merge`: Напишіть рекурсивну функцію `Merge(left_node, right_node)`.
4. Реалізуйте `Insert`: Побудуйте цю операцію на основі `Split` та `Merge`.
5. Реалізуйте `Delete`: Аналогічно, реалізуйте видалення.
6. Протестуйте: Створіть у `main` меню, додайте послідовність елементів, візуалізуйте дерево (або виводьте впорядковано), видаліть декілька елементів та перевірте результат.

6. Результат

Додаток, що реалізує Декартове дерево.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми.
2. Продемонструвати роботу на прикладі, що показує вставку та видалення.
3. Бути готовим відповісти на питання:
    - Пояснити рекурсивну логіку `Split` та `Merge`.
    - Як пріоритет вузла впливає на його положення в дереві?
    - Яка очікувана та найгірша складність вашої реалізації?

Тести

1. Декартове дерево є одночасно:

a) Бінарним деревом пошуку та масивом.

b) Купою та зв’язним списком.

c) Бінарним деревом пошуку та купою .

d) Червоно-чорним деревом та AVL-деревом.

2. Баланс у Декартовому дереві досягається за рахунок:

a) Строгого контролю висот.

b) Випадкових пріоритетів.

c) Кольорів вузлів.

d) Операції `splay`.

3. Яка очікувана часова складність операції вставки в Декартове дерево з N вузлами?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(N \log N)$

4. Операція `Merge(L, R)` коректно працює лише за умови, що:

a) Дерева L та R мають однакову висоту.

b) Всі ключі в L менші за всі ключі в R.

c) Всі пріоритети в L більші за всі пріоритети в R.

d) Дерево R є порожнім.

5. Яка з операцій є фундаментальною для реалізації вставки та видалення в Декартовому дереві?

a) `rightRotate`

b) `fix_color`

c) `Split` та `Merge`

d) `find_height`

Відповіді на прості тести: 1-c, 2-b, 3-b, 4-b, 5-c.

Тести підвищеної складності

1. Структура Декартового дерева для даного набору ключів та пріоритетів є:

a) Залежною від порядку вставки ключів.

b) Завжди однаковою і не залежить від порядку вставки.

c) Залежною від кількості операцій `Split`.

d) Випадковою при кожному запуску програми.

2. Якщо при реалізації операції видалення `Delete(root, key)` ми помилково зробимо `Merge(greater_part, less_part)` замість `Merge(less_part, greater_part)`, що станеться?

a) Нічого, операція `Merge` комутативна.

b) Дерево перетвориться на відсортований список.

c) Буде порушена властивість купи, але властивість BST збережеться.

d) Буде порушена властивість бінарного дерева пошуку.

3. Яка складність побудови Декартового дерева для N елементів, якщо вони надходять вже відсортованими за ключем?

a) $O(N^2)$

b) $O(N\log N)$

c) $O(N)$

d) $O(\log N)$

4. Для якої з наступних операцій Декартове дерево є значно ефективнішим за AVL або червоно-чорне дерево?

a) Пошук елемента.

b) Об’єднання двох дерев (за умови, що ключі одного менші за ключі іншого).

c) Пошук k-ої порядкової статистики.

d) Обхід у відсортованому порядку.

5. У чому полягає основний недолік Декартового дерева порівняно з червоно-чорним деревом?

a) Використовує більше пам’яті.

b) Його продуктивність є імовірнісною, а не гарантованою для кожної окремої операції.

c) Має гіршу асимптотичну складність у середньому випадку.

d) Складніше в реалізації.

6. Якщо всі елементи вставити в Декартове дерево з однаковими пріоритетами, до якої структури воно виродиться?

a) До повного бінарного дерева.

b) До червоно-чорного дерева.

c) До зв’язного списку, що відповідає порядку вставки.

d) До випадкової структури.

7. “Декартове дерево по неявному ключу” (Implicit Treap) використовується для моделювання масиву. Що в ньому виступає в ролі ключа для підтримки властивості BST?

a) Значення елемента.

b) Індекс елемента в масиві.

c) Випадковий пріоритет.

d) Хеш елемента.

8. Яка операція є непотрібною для реалізації `Split` та `Merge` у Декартовому дереві?

a) Рекурсія.

b) Порівняння ключів.

c) Порівняння пріоритетів.

d) Обертання.

Відповіді.

1. *b)* Структура Декартового дерева однозначно визначається набором пар (ключ, пріоритет). Вузол з найбільшим пріоритетом завжди буде коренем, а решта елементів рекурсивно розподіляться по піддеревах. Порядок вставки не має значення.

2. *d)* `Merge` вимагає, щоб усі ключі в лівому аргументі були меншими за ключі в правому. Порушення цієї умови призведе до того, що менші ключі можуть опинитися у правому піддереві, руйнуючи основний інваріант BST.

3. *c)* Якщо елементи надходять відсортованими, кожна вставка відбувається в кінець дерева. Це перетворюється на задачу додавання нового елемента в кінець списку, де на кожному кроці потрібно лише підняти новий вузол вгору по правій гілці відповідно до його пріоритету. Ця операція займає амортизований час $O(1)$, що дає загальну складність $O(N)$.

4. *b)* Операція `Merge` є фундаментальною для Декартового дерева і виконується за $O(\log N)$. В AVL або RBT об'єднання двох дерев є значно складнішою операцією, що вимагає перебудови і має складність $O(N)$ або $O(N \log N)$.

5. *b)* RBT дає жорстку гарантію $O(\log N)$ для кожної операції. Treap дає таку гарантію лише в середньому, "очікувано". Хоча ймовірність отримати погане дерево вкрай мала, вона існує, що робить Treap непридатним для систем жорсткого реального часу.

6. *c)* Якщо пріоритети однакові, операція `Merge` завжди буде обирати один і той самий напрямок (напр., завжди `l` буде коренем). Це призведе до того, що дерево буде будуватися як довгий ланцюжок, тобто виродиться.

7. *b)* У такій структурі ключі не зберігаються явно. Порядок вузлів у дереві (при in-order обході) відповідає їхньому порядку в масиві. Роль ключа для операцій `Split` та `Merge` відіграє індекс (або розмір лівого піддерева), що дозволяє швидко розрізати та об'єднувати масив.

8. *d)* Елегантність підходу на основі **`Split`**/**`Merge`** полягає в тому, що він повністю уникає складних операцій обертання, які є центральними для AVL та RBT. Вся логіка балансування прихована у взаємодії властивостей BST та купи.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте клас Декартового дерева з операціями на основі **`Split`** та **`Merge`**.
2. Ручне трасування. Вручну намалюйте, як буде виглядати Декартове дерево після послідовної вставки пар (ключ, пріоритет): (10, 50), (5, 90), (15, 70), (12, 95).
3. Альтернативна реалізація. Реалізуйте операції вставки та видалення в Декартовому дереві за допомогою обертань, а не **`Split`**/**`Merge`**.
4. Емпіричний аналіз. Проведіть експеримент: побудуйте 1000 різних Декартових дерев для одного й того ж набору з N ключів (з різними випадковими пріоритетами). Обчисліть середню та максимальну висоту отриманих дерев.
5. Пошук k-ої статистики. Модифікуйте вузол Декартового дерева, додавши поле **`size`** (розмір піддерева), та реалізуйте функцію пошуку k-го за величиною елемента за час $O(\log N)$.
6. Об'єднання дерев. Використовуючи вашу реалізацію **`Merge`**, напишіть функцію, що об'єднує два довільні Декартові дерева (ключі можуть перетинатися).
7. Діапазонний запит. Реалізуйте для вашого Декартового дерева функцію, що знаходить усі елементи в заданому діапазоні **`[min, max]`**.
8. Есе-порівняння. Напишіть есе, що порівнює Декартове дерево та Splay-дерево. Що в них спільного (амортизований/імовірнісний аналіз) і в чому ключові відмінності?
9. Декартове дерево по неявному ключу. Дослідіть цю структуру та опишіть, як за її допомогою можна реалізувати операцію **`reverse`** для підмасиву за час $O(\log N)$.
10. Гарантія унікальності пріоритетів. Як зміниться логіка **`Merge`**, якщо пріоритети не є унікальними? Чи виникнуть проблеми?

Дерева з розгалуженням > 2

Досі ми розглядали виключно бінарні дерева, де кожен вузол може мати не більше двох нащадків. Ми бачили, як самозбалансовані бінарні дерева пошуку (AVL, RBT) ефективно вирішують задачу швидкого пошуку та модифікації даних, гарантуючи логарифмічну висоту. Здавалося б, проблему вирішено.

Але що відбувається, коли дані настільки великі, що не вміщуються в оперативну пам'ять (RAM) і зберігаються на повільному носії, як-от SSD або жорсткий диск?

Уявіть, що доступ до RAM — це взяти книгу з полиці у вашій кімнаті, а доступ до диска — це поїздка на інший кінець міста в центральну бібліотеку. Остання операція на тисячі, а то й мільйони разів повільніша. У бінарному дереві кожен перехід від батька до нащадка потенційно є одним таким повільним зверненням до диска. Для дерева з мільярдами елементів, навіть логарифмічна кількість кроків (напр., 30) означатиме 30 окремих, дуже повільних дискових операцій, що є неприйнятним для сучасних баз даних.

Для даних, які зберігаються на диску, високі та "худі" бінарні дерева є неефективними через велику кількість повільних дискових операцій. Рішенням є побудова "низьких" та "широких" дерев, де кожен вузол містить багато ключів і може мати багато нащадків.

Найвідомішою реалізацією цієї ідеї є B-дерево, яке є промисловим стандартом для баз даних та файлових систем. Однак, щоб по-справжньому зрозуміти, як працює складна логіка B-дерева, нам потрібно спершу розглянути його концептуальних попередників. Це простіші структури, які є ідеальними "навчальними моделями" для демонстрації ключової ідеї балансування небінарних дерев — розщеплення вузлів.

2-3-4 дерева

2-3-4 дерева (і їхній менш гнучкий родич, 2-3 дерева) — це найпростіші збалансовані дерева пошуку, які не є бінарними. Їхні правила прості:

- Кожен внутрішній вузол може мати 2, 3 або 4 нащадки.
- Відповідно, кожен вузол зберігає 1, 2 або 3 ключі.
- Найважливіше правило: всі листки знаходяться на однаковій глибині, що гарантує ідеальний баланс дерева.

Баланс підтримується не за допомогою обертань, як в AVL, а за допомогою "росту" вузлів. Коли ми вставляємо новий елемент у вузол, що вже має 3

ключі (і 4 нащадків), він “переповнюється”. Замість складної перебудови, ми просто розщеплюємо цей переповнений 4-ключовий вузол на два простіші 2-ключові вузли, а його середній ключ “виштовхуємо” на рівень вище, до батьківського вузла.

Розуміння цієї операції розщеплення в 2-3-4 дереві є ключем до розуміння B-дерев. B-дерево, по суті, узагальнює цю ідею, дозволяючи вузлам мати не 4, а сотні нащадків.

Узагальнення: (a, b)-дерева.

Щоб формалізувати цю концепцію, в теорії використовується поняття (a,b)-дерева. Це узагальнене дерево пошуку, де кожен внутрішній вузол (окрім кореня) має від **a** до **b** нащадків. B-дерево є лише одним із конкретних типів (a,b)-дерева.

Розуміння цих структур є абсолютно необхідним для будь-кого, хто хоче працювати із системами управління базами даних та високопродуктивними системами зберігання.

У цьому розділі ми:

1. Почнемо з детального розбору 2-3-4 дерев, щоб на простому прикладі зрозуміти фундаментальну логіку вставки та розщеплення вузлів.

2. Покажемо, як ця ідея напряму узагальнюється до B-дерев, які використовуються в реальних системах.

3. Розглянемо найпопулярніші варіації, B-дерева: B-дерево, B+-дерево, B*-дерево, та зрозуміємо, чому саме вони стали стандартом для індексації в базах даних.

Дерева з розгалуженням > 2: 2-3-4 та (a,b)-дерева

1. Теоретичні відомості

1.1. Мотивація: Чому бінарні дерева не завжди ідеальні?

У попередніх розділах ми доклали чимало зусиль, щоб змусити бінарні дерева пошуку бути збалансованими. AVL-дерева використовували строгий контроль висот, червоно-чорні — гнучку систему кольорів, Splay-дерева — опортуністичні обертання. Всі вони вирішували проблему виродження і гарантували логарифмічну складність операцій.

Однак, коли мова заходить про зберігання даних на повільних носіях, як-от жорсткі диски (HDD) або навіть SSD, виникає нова проблема. Доступ до

диска є на тисячі й мільйони разів повільнішим за доступ до оперативної пам'яті. У бінарному дереві, навіть ідеально збалансованому, пошук елемента може вимагати до $\log_2 N$ кроків, і кожен крок потенційно є одним повільним зверненням до диска. Для дерева з мільярдами елементів це може означати десятки дискових операцій, що є неприйнятно повільним.

Ідея вирішення цієї проблеми полягає в тому, щоб зробити дерево значно "нижчим" і "ширшим". Замість того, щоб зчитувати з диска один маленький вузол з одним ключем, ми хочемо за одне звернення зчитувати цілий блок даних, що містить багато ключів. Це приводить нас до ідеї дерев, у яких вузол може мати більше двох нащадків. Найпростішими та найбільш наочними представниками цього класу є 2-3-4 дерева.

1.2. 2-3-4 Дерево: гнучкий вузол та ідеальний баланс

2-3-4 дерево — це збалансоване дерево пошуку, яке підтримує ідеальний баланс (всі листки знаходяться на однаковій глибині) завдяки гнучкій структурі своїх вузлів.

Властивості вузлів:

- 2-вузол (2-node): Має один ключ і двох нащадків (як у звичайному BST).
- 3-вузол (3-node): Має два ключі (K1, K2, де K1 < K2) і трьох нащадків. Лівий нащадок містить ключі менші за K1, середній — між K1 і K2, правий — більші за K2.
- 4-вузол (4-node): Має три ключі (K1, K2, K3) і чотирьох нащадків, організованих за аналогічним принципом.

Головний інваріант: Всі листки дерева (які є порожніми) знаходяться на абсолютно однаковій глибині. Це гарантує, що дерево ніколи не може стати розбалансованим.

Структура вузла:

```
Структура MultiWayNode:
    keys       // Масив ключів (1, 2 або 3 елементи)
    num_keys   // Кількість ключів у вузлі
    children   // Масив вказівників на нащадків (2, 3 або 4 елементи)
    // parent - опціонально
```

Максимальна кількість ключів у 2-3-4 дереві

Щоб знайти максимальну можливу кількість ключів у 2-3-4 дереві висотою h, нам потрібно уявити "ідеально щільне" або максимально заповнене

дерево. У такому дереві кожен вузол (окрім, можливо, на останньому рівні) має максимальну можливу кількість нащадків.

1. Характеристики максимального заповнення

У 2-3-4 дереві максимальну кількість нащадків має 4-вузол.

- 4-вузол містить 3 ключі.
- 4-вузол має 4 нащадків.

Отже, щоб отримати максимальну кількість ключів, ми повинні припустити, що наше дерево висотою **h** складається виключно з 4-вузлів.

2. Структура повного 2-3-4 дерева

Давайте порахуємо кількість вузлів на кожному рівні, починаючи з кореня (рівень 0).

- Рівень 0: 1 вузол (корінь).
- Рівень 1: Оскільки корінь має 4 нащадків, на цьому рівні буде 4 вузли.
- Рівень 2: Кожен з 4 вузлів на попередньому рівні також має 4 нащадків, отже, на цьому рівні буде $4 \times 4 = 16 = 4^2$ вузлів.
- Рівень i: Загалом, на рівні **i** буде 4^i вузлів.

Дерево висотою **h** має **h+1** рівнів (від 0 до **h**).

3. Підрахунок загальної кількості вузлів

Щоб знайти загальну кількість вузлів у такому дереві, нам потрібно просумувати кількість вузлів на всіх рівнях від 0 до **h**. Це сума геометричної прогресії:

Загальна кількість вузлів = $1 + 4 + 4^2 + 4^3 + \ldots + 4^h = \sum_{i=0}^{h} 4^i$

Формула для суми геометричної прогресії: $\sum_{i=0}^{h} r^i = \frac{r^{h+1}-1}{r-1}$. Підставивши наше значення **r=4**, отримуємо:

Загальна кількість вузлів = $\frac{4^{h+1}-1}{4-1} = \frac{4^{h+1}-1}{3}$

4. Підрахунок загальної кількості ключів

Ми встановили, що в нашому максимально заповненому дереві кожен вузол є 4-вузлом і містить рівно 3 ключі. Щоб знайти загальну кількість ключів, потрібно помножити загальну кількість вузлів на 3:

Загальна кількість ключів = (Загальна кількість вузлів) × 3 Загальна кількість ключів = $(\frac{4^{h+1}-1}{3}) \times 3 = 4^{h+1} - 1$

Приклад: Розглянемо дерево висотою **h=1**.

- Воно має 2 рівні (0 та 1).
- На рівні 0: 1 вузол (корінь) з 3 ключами.
- На рівні 1: 4 вузли (листки), кожен з 3 ключами.
- Загальна кількість ключів = $3 + (4 \times 3) = 3 + 12 = 15$.
- Перевіримо за формулою: $4^{1+1} - 1 = 4^2 - 1 = 16 - 1 = 15$.

Формула працює коректно.

Операція пошуку

Пошук у 2-3-4 дереві є простим узагальненням пошуку в BST.

```
ФУНКЦІЯ Search(node, key):
    ЯКЩО node == NULL ТО ПОВЕРНУТИ false КІНЕЦЬ ЯКЩО

    // Шукаємо правильний діапазон серед ключів поточного вузла
    // і визначаємо, в яке піддерево спускатися
    i = 0
    ПОКИ i < node.num_keys І key > node.keys[i]:
        i = i + 1
    КІНЕЦЬ ПОКИ

    // Якщо знайшли ключ у поточному вузлі
    ЯКЩО i < node.num_keys І key == node.keys[i] ТО
        ПОВЕРНУТИ true
    КІНЕЦЬ ЯКЩО

    // Якщо це листок, і ми не знайшли ключ, його немає
    ЯКЩО isLeaf(node) ТО
        ПОВЕРНУТИ false
    КІНЕЦЬ ЯКЩО

    // Інакше, рекурсивно спускаємось у відповідне піддерево
    ПОВЕРНУТИ Search(node.children[i], key)
КІНЕЦЬ ФУНКЦІЇ
```

Операція вставки: проактивне розщеплення

Це серце алгоритму. На відміну від AVL-дерев, де ми спершу вставляємо, а потім “ремонтуємо” дерево на шляху вгору, 2-3-4 дерево діє проактивно.

Ідея: Ми спускаємося по дереву в пошуках місця для вставки. Якщо на своєму шляху ми зустрічаємо 4-вузол (переповнений вузол), ми негайно розщеплюємо його, не чекаючи на повернення.

Процедура розщеплення 4-вузла: Уявімо 4-вузол `X` з ключами `{L, M, R}` та чотирма нащадками `{c1, c2, c3, c4}`.

1. Середній ключ `M` “виштовхується” вгору, до батьківського вузла `P`.
2. Вузол `X` розщеплюється на два нових 2-вузли:
    - Лівий вузол `X_left` отримує ключ `L` та нащадків `c1`, `c2`.
    - Правий вузол `X_right` отримує ключ `R` та нащадків `c3`, `c4`.
3. Ці два нові вузли стають нащадками батьківського вузла `P`, приєднуючись до ключа `M`, що піднявся.

Особливий випадок: розщеплення кореня. Якщо 4-вузол, який ми розщеплюємо, є коренем, у нього немає батька. В цьому випадку середній ключ `M` стає новим коренем дерева, а два нових 2-вузли стають його нащадками. Саме так дерево росте у висоту.

Псевдокод вставки:

```
ФУНКЦІЯ Insert(root, key):
    // Якщо корінь є 4-вузлом, розщеплюємо його одразу.
    // Це єдиний випадок, коли висота дерева може зрости.
    ЯКЩО root.num_keys == 3 ТО
        new_root = Створити_Вузол()
        new_root.children[0] = root
        splitChild(new_root, 0) // Функція розщеплення
        root = new_root
    КІНЕЦЬ ЯКЩО

    insertNonFull(root, key)
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ insertNonFull(node, key):
    // Якщо це листок, просто вставляємо ключ
    ЯКЩО isLeaf(node) ТО
        // ... логіка вставки ключа у відсортований масив ключів вузл
а ...
        node.num_keys = node.num_keys + 1
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // Знаходимо, в яке піддерево спускатися
    i = знайти_індекс_піддерева(node, key)

    // Якщо нащадок - 4-вузол, розщеплюємо його
    ЯКЩО node.children[i].num_keys == 3 ТО
        splitChild(node, i)
        // Після розщеплення потрібно перевірити, чи не змінився шлях
```

```
        ЯКЩО key > node.keys[i] ТО
            i = i + 1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ЯКЩО

    // Рекурсивно спускаємось у відповідне піддерево
    insertNonFull(node.children[i], key)
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ splitChild(parent_node, child_index):
    // ... детальна логіка розщеплення 4-вузла parent_node.children[c
hild_index]
    // та підняття середнього ключа в parent_node ...
КІНЕЦЬ ФУНКЦІЇ
```

Операція видалення

Видалення у 2-3-4 дереві є складнішим. Основна ідея — звести задачу до видалення з листка.

1. Пошук: Знаходимо ключ K, який потрібно видалити.
2. Якщо K знаходиться у внутрішньому вузлі: Міняємо його місцями з його наступником (in-order successor), який гарантовано знаходиться в листку. Тепер задача звелася до видалення K з листка.
3. Видалення з листка: Просто видаляємо ключ.
4. Виправлення "недоповнення": Якщо після видалення вузол стає "нелегальним" (напр., 1-вузлом, що не є коренем), потрібно відновити інваріант. Це робиться за допомогою запозичення ключа у "багатого" сусіднього вузла або шляхом злиття з ним. Цей процес може розповсюджуватися вгору по дереву.

1.5. 2-3 Дерева та (a, b)-дерева

– 2-3 Дерево: Це просто підвид 2-3-4 дерева, в якому 4-вузли заборонені. Логіка вставки трохи інша: ми спускаємося до листка, вставляємо ключ, і якщо вузол стає 4-вузлом, ми розщеплюємо його і "проштовхуємо" проблему вгору до батька (bottom-up підхід, на відміну від top-down у 2-3-4 деревах).
– (a, b)-Дерево: Це формальне узагальнення. Це збалансоване дерево пошуку, де кожен внутрішній вузол (окрім кореня) має від a до b нащадків.
    - a — мінімальний ступінь розгалуження ($a \geq 2$).

- b — максимальний ступінь розгалуження ($b \geq 2a - 1$).
- 2-3-4 дерево є (2, 4)-деревом.
- 2-3 дерево є (2, 3)-деревом.
- B-дерево є найвідомішим представником (a,b)-дерев.

Аналіз складності операцій у 2-3-4 та (a,b)-деревах

Фундаментальна перевага 2-3-4 та, узагальнено, (a,b)-дерев полягає в тому, що вони завжди залишаються збалансованими. На відміну від простого BST, тут не існує "найгіршого випадку" у вигляді виродженого дерева. Це досягається завдяки механізмам розщеплення та злиття вузлів, які гарантують, що висота дерева h для N елементів завжди є логарифмічною.

1. **Аналіз висоти дерева**

Давайте зрозуміємо, чому висота є логарифмічною. У 2-3-4 дереві кожен внутрішній вузол має щонайменше 2 нащадки. Це означає, що на кожному наступному рівні кількість вузлів щонайменше подвоюється. Для дерева з N елементів, висота h буде обмежена величиною $O(\log_2 N)$.

Для загального (a,b)-дерева, де кожен вузол має мінімум a нащадків ($a \geq 2$), висота буде обмежена величиною $O(\log_a N)$. З точки зору асимптотичного аналізу, основа логарифма не має значення, оскільки логарифми з різними основами відрізняються лише на константний множник. Тому для всіх цих дерев ми можемо стверджувати, що їхня висота $h = O(\log N)$.

Цей факт є наріжним каменем для аналізу всіх операцій.

2. **Пошук**

– Найгірший випадок: $O(\log N)$ Процес пошуку починається з кореня і на кожному рівні спускається на один крок глибше. Кількість вузлів, які нам потрібно відвідати на шляху від кореня до листка, дорівнює висоті дерева, тобто $O(\log N)$. Всередині кожного вузла нам потрібно знайти правильний діапазон серед його ключів, щоб обрати наступний шлях. Оскільки кількість ключів у вузлі є обмеженою константою (максимум 3 для 2-3-4 дерева або b-*1* для (a,b)-дерева), пошук всередині вузла можна виконати за константний час $O(1)$ (для малих b) або за $O(\log b)$ (якщо використовувати бінарний пошук по ключах). Загальна складність є добутком висоти на вартість обробки вузла: $O(\log N) \times O(\log b) = O(\log N)$, оскільки b є константою, не залежною від N.

– Середній випадок: $O(\log N)$ Оскільки дерево завжди збалансоване, середня довжина шляху також є логарифмічною.

3. Вставка

– Найгірший випадок: $O(\log N)$ Аналіз для 2-3-4 дерева з проактивним розщепленням "зверху-вниз":

1. Спуск по дереву: Ми проходимо один шлях від кореня до листка. Це займає $O(h) = O(\log N)$ кроків.
2. Розщеплення на шляху: На кожному кроці ми можемо натрапити на 4-вузол і виконати його розщеплення. Операція `splitChild` є локальною і вимагає константної кількості операцій з масивами та вказівниками, тому її складність — $O(1)$.
3. Вставка в листок: Вставка ключа у масив ключів листка також займає $O(1)$, оскільки розмір масиву обмежений.

Загальна складність є добутком кількості кроків на вартість кожного кроку: $O(\log N) \times O(1) = O(\log N)$.

Для 2-3 дерева з підходом "знизу-вгору" логіка схожа: спуск $O(\log N)$, а потім потенційний підйом з розщепленнями, який також обмежений висотою дерева, $O(\log N)$.

– Середній випадок: $O(\log N)$ Структура гарантує збалансованість, тому середня продуктивність не відрізняється від найгіршої.

4. Видалення

– Найгірший випадок: $O(\log N)$ Видалення є найскладнішою операцією, але її складність також обмежена висотою дерева.

4. Пошук елемента: Перший крок — знайти вузол, що містить ключ для видалення. Це займає $O(\log N)$.
5. Заміна (за потреби): Якщо ключ знаходиться не в листку, ми міняємо його місцями з його наступником, який гарантовано знаходиться в листку. Пошук наступника також вимагає спуску по дереву, що займає $O(\log N)$.
6. Видалення з листка: Сама операція видалення ключа з масиву у вузлі — $O(1)$.
7. Балансування ("знизу-вгору"): Після видалення листок може стати "недоповненим". Ми піднімаємось вгору по дереву до кореня, виконуючи операції запозичення або злиття. Кожна така

операція є локальною і має складність $O(1)$. У найгіршому випадку цей процес може дійти до самого кореня, вимагаючи $O(h) = O(\log N)$ операцій.

Сумарна складність всіх цих етапів є логарифмічною: $O(\log N) + O(\log N) + O(\log N) = O(\log N)$.

– Середній випадок: $O(\log N)$ Аналогічно до попередніх операцій.

5. Обходи дерева

– Часова складність: $O(N)$ Для будь-якого повного обходу (симетричного, прямого, зворотного, по рівнях) нам потрібно відвідати кожен ключ у дереві рівно один раз. Якщо в дереві N ключів, то і складність буде лінійною відносно N.

– Просторова складність:

- DFS (рекурсивні обходи In-order, Pre-order, Post-order): Визначається максимальною глибиною стека рекурсії, яка дорівнює висоті дерева. Для збалансованого (a,b)-дерева це $O(\log N)$.
- BFS (обхід в ширину Level-order): Визначається максимальним розміром черги. У найгіршому випадку (для широкого дерева) в черзі може опинитися велика кількість вузлів. Просторова складність — $O(N)$.

2. Область застосування

– Педагогічні цілі: 2-3 та 2-3-4 дерева є ідеальними для вивчення та розуміння принципів роботи небінарних збалансованих дерев.

– Концептуальна основа: Вони є прямою основою для B-дерев, які використовуються в:

- Системах управління базами даних (СУБД): для індексації (напр., Oracle, PostgreSQL).
- Файлових системах: для організації структури каталогів та файлів (напр., NTFS, HFS+).

3. Приклади застосування

– Ізоморфізм з червоно-чорними деревами: Будь-яке червоно-чорне дерево можна представити як 2-3-4 дерево, де червоний вузол “об’єднується” зі своїм чорним батьком, утворюючи 3- або 4-вузол.

Розуміння 2-3-4 дерев дає глибше уявлення про те, чому правила RBT гарантують баланс.

Переваги та недоліки

Переваги:

- Ідеальний баланс: 2-3-4 дерева завжди залишаються ідеально збалансованими по висоті, що гарантує логарифмічну складність $O(\log n)$ для всіх операцій у найгіршому випадку.
- Менша висота: За рахунок більшого розгалуження, висота 2-3-4 дерева є меншою, ніж у бінарного дерева з тією ж кількістю елементів, що зменшує кількість кроків при пошуку.

Недоліки:

- Складніша структура вузла: Вузол має зберігати масив ключів та масив вказівників, що ускладнює реалізацію та вимагає більше пам’яті на один вузол.
- Більше порівнянь у вузлі: На кожному кроці пошуку потрібно виконати не одне, а декілька порівнянь всередині вузла, щоб знайти правильний шлях. Для даних, що зберігаються в оперативній пам’яті, це може нівелювати виграш від меншої висоти.

Контрольні запитання

1. У чому полягає головна відмінність вузла 2-3-4 дерева від вузла бінарного дерева?
2. Опишіть основну властивість 2-3-4 дерева, яка гарантує його баланс.
3. Що таке “проактивне розщеплення” і чому воно використовується при вставці?
4. Опишіть покроково, що відбувається при розщепленні 4-вузла.
5. Як 2-3-4 дерева пов’язані з B-деревами та червоно-чорними деревами?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація 2-3-4 дерева та операції вставки.

2. Мета

Практичне засвоєння логіки роботи небінарних збалансованих дерев. Реалізація механізму проактивного розщеплення вузлів для підтримки балансу.

3. Завдання

Створити програму, що реалізує 2-3-4 дерево та операцію вставки. Продемонструвати, як дерево росте у висоту та як відбувається розщеплення вузлів.

4. Опис

Програма має складатися з класу/структури `MultiWayNode` та основного класу `Tree234`. Вузол має містити масиви для ключів та вказівників на нащадків.

5. Порядок виконання

1. Створіть клас `MultiWayNode`.
2. Реалізуйте `insert(key)`: Це головна функція, яка перевіряє корінь на переповнення і викликає рекурсивного помічника.
3. Реалізуйте `insertNonFull(node, key)`: Рекурсивна функція, що спускається по дереву, розщеплюючи 4-вузли на шляху.
4. Реалізуйте `splitChild(...)`: Це найскладніша частина, що реалізує логіку розщеплення вузла.
5. Реалізуйте обхід: Додайте метод для симетричного обходу, щоб перевіряти коректність роботи.
6. Протестуйте: Вставляйте послідовно числа (напр., 10, 20, 30, 5, 15, 25, 35) і після кожної вставки виводьте структуру дерева, щоб бачити, як і коли відбуваються розщеплення.

6. Результат

Додаток, що реалізує 2-3-4 дерево та коректно підтримує його інваріанти при вставці.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми.
2. Продемонструвати роботу додатку, показавши сценарій, що призводить до розщеплення кореня.
3. Бути готовим відповісти на питання:
    - Пояснити, як працює ваша функція `splitChild`.

- Чому ваш алгоритм перевіряє на переповнення вузли на шляху вниз, а не вгору?
- Яка гарантована часова складність вашої операції вставки?

Тести

1. Вузол у 2-3-4 дереві НЕ може мати:

a) 1 ключ та 2 нащадки.

b) 2 ключі та 3 нащадки.

c) 3 ключі та 4 нащадки.

d) 4 ключі та 5 нащадків.

2. Яка властивість гарантує, що 2-3-4 дерево завжди збалансоване?

a) Кожен вузол має не більше 4 нащадків.

b) Всі листки знаходяться на однаковій глибині.

c) Використовуються обертання для балансування.

d) Всі ключі у вузлі відсортовані.

3. Що відбувається, коли при вставці “зверху-вниз” ми зустрічаємо 4-вузол?

a) Ми продовжуємо спускатися, ігноруючи його.

b) Вставка завершується з помилкою.

c) Ми негайно розщеплюємо цей 4-вузол, перш ніж спускатися далі.

d) Ми виконуємо ліве обертання.

4. 2-3-4 дерево є конкретним випадком:

a) AVL-дерева.

b) Червоно-чорного дерева.

c) (a, b)-дерева.

d) Splay-дерева.

5. Основна перевага 2-3-4 дерева над бінарним деревом пошуку полягає у:

a) Меншому використанні пам’яті.

b) Простішій реалізації пошуку.

c) Гарантованій логарифмічній висоті у найгіршому випадку.

d) Швидшій операції обходу.

Відповіді на прості тести: 1-d, 2-b, 3-c, 4-c, 5-c.

Тести підвищеної складності

1. Після якої з наступних послідовностей вставок у порожнє 2-3-4 дерево його висота стане рівною 1 (тобто, з'явиться другий рівень)?

a) 10, 20

b) 10, 20, 30

c) 10, 20, 30, 40

d) 10

2. Чим реалізація вставки у 2-3 дереві (де 4-вузли заборонені) відрізняється від 2-3-4 дерева?

a) Вона використовує обертання замість розщеплень.

b) Вона є нерекурсивною.

c) Розщеплення виконується "знизу-вгору" (bottom-up): вузол розщеплюється лише після того, як вставка в нього призвела до переповнення, і проблема може поширюватися до кореня.

d) Жодних відмінностей немає.

3. Яка структура даних є ізоморфною до 2-3-4 дерева?

a) AVL-дерево.

b) Splay-дерево.

c) Декартове дерево.

d) Червоно-чорне дерево.

4. Якщо видалення ключа з листка 2-3-4 дерева призвело до того, що листок став 1-вузлом (містить 0 ключів), яка перша дія для виправлення?

a) Видалити цей вузол.

b) Спробувати "запозичити" ключ у сусіднього брата, якщо той має більше одного ключа.

c) Негайно виконати злиття з братом.

d) Перебудувати все дерево заново.

5. У порівнянні з AVL-деревом, 2-3-4 дерево:

a) Виконує менше порівнянь, але більше структурних змін при пошуку.

b) Має меншу висоту та вимагає менше дискових доступів для даних, що зберігаються на диску.

c) Є значно швидшим для даних, що зберігаються в оперативній пам'яті.

d) Має гіршу асимптотичну складність у найгіршому випадку.

6. Яка максимальна кількість ключів може бути у 2-3-4 дереві висотою `h`?

a) $4^h - 1$

b) $3^h - 1$

c) $4^{h+1} - 1$

d) $3^{h+1} - 1$

7. У чому полягає основний недолік "bottom-up" вставки (як у 2-3 дереві) порівняно з "top-down" (як у 2-3-4 дереві)?

a) Вона повільніша асимптотично.

b) Вона вимагає зберігання вказівників на батьківські вузли або використання стека для повернення назад, що ускладнює реалізацію.

c) Вона призводить до більш розбалансованих дерев.

d) Вона не підтримує видалення.

8. Який з інваріантів червоно-чорного дерева напряму відповідає правилу 2-3-4 дерева, що всі листки на одній глибині?

a) Правило кореня (корінь чорний).

b) Правило червоних вузлів (червоний вузол не може мати червоного нащадка).

c) Правило чорної висоти (всі шляхи від вузла до листків містять однакову кількість чорних вузлів).

d) Жоден з них.

Відповіді.

1. *b)* Після вставки 10 і 20 ми маємо один 3-вузол `{10, 20}`. Після вставки 30 ми отримуємо 4-вузол `{10, 20, 30}`. Оскільки це корінь, він розщеплюється: 20 стає новим коренем, а 10 і 30 — його нащадками. Висота стає 1.

2. *c)* Оскільки в 2-3 дереві немає "запасної" комірки, як у 2-3-4, ми не можемо проактивно розщеплювати. Ми вставляємо в листок, і якщо він

переповнюється, ми розщеплюємо його і передаємо середній ключ батькові, що може викликати ланцюгову реакцію розщеплень вгору.

3. *d)* Існує пряма відповідність. 2-вузол відповідає чорному вузлу. 3-вузол — чорному вузлу з одним червоним нащадком. 4-вузол — чорному вузлу з двома червоними нащадками. Операції розщеплення та зміни кольорів є еквівалентними.

4. *b)* Запозичення є “дешевшою” операцією, ніж злиття, оскільки воно не зменшує кількість вузлів на рівні. Злиття виконується лише тоді, коли всі сусідні брати також є “бідними” 2-вузлами і не можуть нічим поділитися.

5. *b)* Головна ідея мульти-дерев — зменшення висоти. Для даних на диску це означає менше повільних операцій I/O. Для даних в RAM, через більшу кількість порівнянь у вузлі, 2-3-4 дерево може бути навіть повільнішим за AVL.

6. *c)* Пояснення дивись у підручнику

7. *b)* “Top-down” розщеплення гарантує, що коли ми спускаємось, у батьківського вузла завжди є місце для нового ключа. “Bottom-up” розщеплення може викликати ланцюгову реакцію, що йде вгору до кореня, і для цього потрібно пам’ятати пройдений шлях.

8. *c)* Чорна висота є прямим аналогом висоти в 2-3-4 дереві. Правило про однакову кількість чорних вузлів гарантує, що дерево залишається збалансованим, так само як правило про однакову глибину листків у 2-3-4 дереві.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте клас 2-3-4 дерева з операцією вставки.
2. Ручне трасування. Вручну намалюйте, як буде змінюватися 2-3-4 дерево при послідовній вставці ключів: 1, 2, 3, 4, 5, 6, 7. Покажіть усі розщеплення.
3. Реалізація видалення. *(Завдання підвищеної складності)*. Додайте до вашого класу метод `delete`, що підтримує балансування через запозичення та злиття.
4. Аналіз висоти. Напишіть функцію, що обчислює висоту 2-3-4 дерева. Порівняйте її з висотою звичайного BST для однакових наборів даних.

5. 2-3 Дерево. Модифікуйте вашу реалізацію так, щоб вона реалізовувала 2-3 дерево з “bottom-up” вставкою.
6. Порівняння підходів. Напишіть есе, що порівнює “top-down” та “bottom-up” підходи до вставки у 2-3-4 та 2-3 деревах.
7. Перетворення в RBT. Напишіть функцію, що перетворює задане 2-3-4 дерево на еквівалентне червоно-чорне дерево.
8. Об’єднання дерев. Дослідіть, як можна реалізувати операцію `Merge` для двох 2-3-4 дерев.
9. Пошук k-ої статистики. Модифікуйте вузол 2-3-4 дерева, щоб він зберігав розмір піддерева, та реалізуйте пошук k-го за величиною елемента.

B-дерева: B-дерево, B+-дерево, B*-дерево

Узагальнені дерева пошуку: B-дерева та їхні варіації

1. Теоретичні відомості

1.1. Мотивація: тиранія дискової пам’яті

Досі ми розглядали структури даних, неявно припускаючи, що всі вони знаходяться у швидкій оперативній пам’яті (RAM). Ми бачили, що збалансовані бінарні дерева, як-от AVL та червоно-чорні, забезпечують чудову логарифмічну складність $O(\log N)$ для основних операцій. Для мільйона елементів це означає близько 20 кроків-порівнянь, що є фантастичним результатом.

Але що відбувається, коли даних стає настільки багато, що вони не вміщуються в RAM і мають зберігатися на значно повільніших носіях — SSD або, тим більше, жорстких дисках (HDD)?

У цьому випадку правила гри кардинально змінюються. Доступ до диска — це надзвичайно повільна операція. Час, потрібний для одного зчитування блоку з диска, вимірюється в мілісекундах, тоді як операції в RAM — у наносекундах. Різниця складає тисячі, а то й мільйони разів.

У бінарному дереві кожен перехід від батька до нащадка потенційно є однією такою повільною операцією читання з диска. Для дерева з мільярдами елементів навіть логарифмічна кількість кроків (напр., 30) означатиме 30 окремих, дуже повільних дискових операцій. Для сучасних

баз даних, які мають обробляти тисячі запитів на секунду, такий підхід є абсолютно неприйнятним.

Проблема полягає у “високих” та “худих” бінарних деревах. Нам потрібна структура, яка є, навпаки, надзвичайно “низькою” та “широкою”.

1.2. Ідея B-дерева: “товсті” вузли

B-дерево — це збалансоване дерево пошуку, розроблене спеціально для ефективної роботи з даними, що зберігаються на блокових пристроях (як диски). Його ключова ідея полягає в тому, щоб мінімізувати кількість дискових доступів.

Як це досягається? Замість того, щоб зчитувати з диска один маленький вузол з одним ключем, ми за одне звернення зчитуємо цілий блок пам’яті (який називають сторінкою або вузлом), розмір якого зазвичай дорівнює розміру сторінки файлової системи (напр., 4 КБ). Цей блок містить не один, а десятки або сотні ключів.

Вузол у B-дереві — це не просто контейнер для одного ключа, а ціла відсортована “сторінка”, що може містити багато ключів та, відповідно, багато вказівників на нащадків. Завдяки цьому дерево стає надзвичайно “низьким” та “широким”. Його висота зростає набагато повільніше, ніж у бінарного дерева. Для бази даних з трильйоном записів висота B-дерева може складати всього 4-5 рівнів! Це означає, що для знаходження будь-якого запису потрібно лише 4-5 дискових операцій, що є величезним виграшем у продуктивності.

1.3. Сімейство B-дерев

Хоча ідея “товстих” вузлів є спільною, існують різні варіації, кожна з яких оптимізована для певних завдань. Ми розглянемо три основні:

1. Класичне B-дерево: Фундаментальна структура, яка є узагальненням 2-3-4 дерева. Вона ефективна, але має певні недоліки для специфічних типів запитів.
2. B+-дерево: Найпопулярніша варіація, яка стала промисловим стандартом для індексів у реляційних базах даних. Вона вводить дві ключові зміни: зберігає дані лише в листках і зв’язує листки у список для надзвичайно ефективних діапазонних запитів.
3. B*-дерево: Модифікація, сфокусована на максимально щільному використанні дискового простору. Вона вимагає, щоб вузли були

заповнені щонайменше на 2/3, що зменшує частоту розщеплень і робить дерево ще більш компактним.

1.4. Загальні властивості та операції

Хоча реалізації відрізняються, загальна логіка операцій є схожою. Пізніше ми розглянемо специфіку цих операцій для кожного різновиду дерев окремо.

Структура вузла B-дерева (узагальнена):

```
Структура BTreeNode:
    keys[]     // Масив ключів
    children[] // Масив вказівників на нащадків
    num_keys   // Поточна кількість ключів у вузлі
    is_leaf    // Прапорець, чи є вузол листком
```

Операція пошуку (узагальнена):

Пошук завжди починається з кореня і рекурсивно спускається по дереву. На кожному вузлі ми шукаємо потрібний діапазон серед ключів, щоб визначити, в яке піддерево рухатись далі.

```
ФУНКЦІЯ Search(node, key):
    // 1. Знаходимо індекс 'i' такий, що key <= node.keys[i]
    i = 0
    ПОКИ i < node.num_keys І key > node.keys[i]:
        i = i + 1
    КІНЕЦЬ ПОКИ

    // 2. Якщо ключ знайдено у поточному вузлі
    ЯКЩО i < node.num_keys І key == node.keys[i] ТО
        ПОВЕРНУТИ (node, i) // Знайдено
    КІНЕЦЬ ЯКЩО

    // 3. Якщо це листок, і ми не знайшли ключ
    ЯКЩО node.is_leaf == true ТО
        ПОВЕРНУТИ NULL // Не знайдено
    КІНЕЦЬ ЯКЩО

    // 4. Інакше, рекурсивно спускаємось у відповідне піддерево
    // Завантажуємо з диска вузол node.children[i]
    ПОВЕРНУТИ Search(node.children[i], key)
КІНЕЦЬ ФУНКЦІЇ
```

Операції вставки та видалення (узагальнена)**:**

Вони базуються на тих самих принципах, що й у 2-3-4 деревах:

– Вставка: Якщо вузол переповнюється, він розщеплюється на два, а середній ключ “виштовхується” вгору до батька.
– Видалення: Якщо вузол стає “недоповненим”, він намагається запозичити ключ у сусіда або зливається з ним.

У наступних розділах ми детально розглянемо, як саме ці операції реалізовані для кожного типу B-дерева.

2. Область застосування

– Системи управління базами даних (СУБД): Це основна сфера. B+-дерева є стандартом для побудови індексів, що дозволяють швидко знаходити записи за ключем.
– Файлові системи: NTFS, HFS+, ext4 та інші використовують варіації B-дерев для організації метаданих файлів, каталогів та відстеження вільних блоків на диску.

3. Приклади застосування

– MySQL (InnoDB): Індекси таблиць реалізовані за допомогою B+-дерев.
– PostgreSQL: Аналогічно, використовує B+-дерева для стандартних індексів.
– MongoDB (WiredTiger): Використовує B-дерева (не B+) для своєї внутрішньої структури.

Розглянемо більш детально виконання всіх операцій для кожного дерева окремо.

Операції у класичному B-дереві

Реалізація операцій у B-дереві є втіленням інженерного мистецтва, спрямованого на мінімізацію дискових доступів. Кожна операція спроєктована так, щоб виконувати якомога більше роботи з вузлом (сторінкою), який вже завантажено в оперативну пам’ять, перш ніж звертатися до диска за наступним.

Операція Пошуку

Пошук у B-дереві — це прямолінійний процес спуску від кореня до цільового вузла.

Процедура:

1. Починаємо з кореня. Завантажуємо кореневий вузол `node` в пам’ять.

2. Пошук у поточному вузлі. Виконуємо пошук шуканого ключа `key` серед ключів, що вже знаходяться у вузлі `node`. Оскільки ключі у вузлі впорядковані, для цього можна використати ефективний бінарний пошук (або лінійний, якщо кількість ключів мала).
3. Аналіз результату пошуку у вузлі:
    - Якщо ключ `key` знайдено у поточному вузлі на позиції `i`, це означає успіх. Ми повертаємо цей вузол та індекс `i`. Операція завершена.
    - Якщо ключ `key` не знайдено, пошук у вузлі зупиниться на позиції `i`, де `node.keys[i-1] < key < node.keys[i]`. Це означає, що наш ключ, якщо він існує, може знаходитись лише в піддереві, на яке вказує `node.children[i]`.
4. Рекурсивний спуск або завершення:
    - Якщо поточний вузол `node` є листком (`is_leaf == true`), і ми не знайшли ключ, це означає, що його в дереві немає. Пошук завершується невдачею.
    - Якщо `node` є внутрішнім вузлом, ми завантажуємо з диска дочірній вузол, на який вказує `node.children[i]`, і повторюємо процедуру з кроку 2 для цього нового вузла.

Псевдокод такий:

```
// Вхід: node - поточний вузол, key - ключ для пошуку.
// Вихід: пара (вузол, індекс), де знайдено ключ, або NULL.
ФУНКЦІЯ Search(node, key):
    // 1. Знаходимо індекс першого ключа, що більший або рівний нашому.
    i = 0
    ПОКИ i < node.num_keys І key > node.keys[i]:
        i = i + 1
    КІНЕЦЬ ПОКИ

    // 2. Перевіряємо, чи ми знайшли точний збіг.
    ЯКЩО i < node.num_keys І key == node.keys[i] ТО
        ПОВЕРНУТИ (node, i) // Успіх!
    КІНЕЦЬ ЯКЩО

    // 3. Якщо це листок, шлях пошуку завершено.
    ЯКЩО node.is_leaf == true ТО
        ПОВЕРНУТИ NULL // Не знайдено
```

```
    КІНЕЦЬ ЯКЩО

    // 4. Інакше, завантажуємо дочірній вузол і продовжуємо пошук.
    // DISK_READ(node.children[i]) - імітує операцію читання з диска.
    child_node = DISK_READ(node.children[i])
    ПОВЕРНУТИ Search(child_node, key)
КІНЕЦЬ ФУНКЦІЇ
```

Операція Вставки

Вставка в B-дерево використовує стратегію проактивного розщеплення. Щоб уникнути складної логіки повернення вгору по дереву, ми на шляху вниз гарантуємо, що вузол, в який ми збираємось спуститися, не є переповненим.

Процедура:

1. Спуск та розщеплення. Ми починаємо рух від кореня до відповідного листка, куди потрібно вставити новий ключ. На кожному кроці, перш ніж перейти до дочірнього вузла, ми перевіряємо, чи цей дочірній вузол переповнений (тобто, містить **`2t-1`** ключів).
    - Якщо нащадок переповнений, ми негайно розщеплюємо його. Процедура **`splitChild`** розбиває переповнений дочірній вузол на два нових, а його середній ключ “виштовхується” вгору, до поточного батьківського вузла. Після цього ми визначаємо, в який із двох новоутворених нащадків нам потрібно спуститися.
    - Якщо нащадок не переповнений, ми просто продовжуємо спуск.
2. Вставка в листок. Цей проактивний підхід гарантує, що коли ми нарешті дійдемо до листкового вузла, в ньому завжди буде вільне місце. Ми просто вставляємо новий ключ у масив ключів цього листка, зберігаючи їхній відсортований порядок.

Особливий випадок: розщеплення кореня. Якщо корінь переповнюється, він розщеплюється. Його середній ключ стає новим коренем, який має двох нащадків. Це єдиний випадок, коли висота B-дерева збільшується.

Псевдокод такий:

```
// Основна функція вставки
ФУНКЦІЯ Insert(root, key):
    r = root
    // Якщо корінь переповнений, створюємо новий корінь і розщеплюємо старий.
    ЯКЩО r.num_keys == 2*t - 1 ТО
```

```
        s = Створити_Вузол(is_leaf=false)
        s.children[0] = r
        splitChild(s, 0) // Розщеплюємо r як нащадка нового кореня s
        root = s // s - тепер новий корінь
        insertNonFull(root, key)
    ІНАКШЕ
        insertNonFull(r, key)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ

// Рекурсивна функція вставки у непереповнений вузол
ФУНКЦІЯ insertNonFull(node, key):
    i = node.num_keys - 1
    // Якщо це листок, знаходимо місце і вставляємо ключ
    ЯКЩО node.is_leaf == true ТО
        ПОКИ i >= 0 І key < node.keys[i]:
            node.keys[i + 1] = node.keys[i]
            i = i - 1
        КІНЕЦЬ ПОКИ
        node.keys[i + 1] = key
        node.num_keys = node.num_keys + 1
    // Якщо це внутрішній вузол
    ІНАКШЕ:
        // Знаходимо індекс нащадка для спуску
        ПОКИ i >= 0 І key < node.keys[i]:
            i = i - 1
        КІНЕЦЬ ПОКИ
        i = i + 1

        // Завантажуємо нащадка з диска
        child_node = DISK_READ(node.children[i])
        // Якщо він переповнений, розщеплюємо його
        ЯКЩО child_node.num_keys == 2*t - 1 ТО
            splitChild(node, i)
            // Визначаємо, в якого з двох нових братів тепер потрібно
йти
            ЯКЩО key > node.keys[i] ТО
                i = i + 1
            КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ЯКЩО
        // Рекурсивно викликаємо для відповідного нащадка
        insertNonFull(node.children[i], key)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

```
// Функція розщеплення переповненого нащадка
ФУНКЦІЯ splitChild(parent_node, child_index):
    // ... детальна логіка розщеплення вузла parent_node.children[chi
ld_index]
    // на два нових, та підняття середнього ключа в parent_node.
    // Ця логіка аналогічна до B+-дерева, але тут також копіюються
    // вказівники на нащадків.
КІНЕЦЬ ФУНКЦІЇ
```

Операція Видалення

Видалення є найскладнішою операцією, оскільки вимагає підтримки інваріанту про мінімальну кількість ключів (`t-1`) у вузлі. На відміну від вставки, тут використовується "реактивний" підхід: ми спершу видаляємо, а потім виправляємо можливі проблеми.

Процедура:

1. Пошук ключа `key` у дереві.
2. Сценарій 1: Ключ знаходиться у листковому вузлі `L`.
    - Просто видаляємо ключ з масиву ключів вузла `L`. Якщо після цього `L` все ще має щонайменше `t-1` ключів, операція завершена. Якщо ні, запускається процедура відновлення.
3. Сценарій 2: Ключ знаходиться у внутрішньому вузлі `V`.

    Ми не можемо просто видалити ключ, оскільки це порушить структуру зв'язків.

    a) Знаходимо заміну: Ми знаходимо для `key` його наступника (in-order successor), який гарантовано знаходиться в листковому вузлі.

    c) Міняємо місцями: Міняємо `key` місцями з його наступником.

    d) Зводимо до сценарію 1: Тепер наше завдання звелось до видалення ключа з листкового вузла, що ми вже вміємо робити.
4. Процедура відновлення (після видалення з листка, що став "недоповненим"): Коли вузол `L` має менше `t-1` ключів, ми дивимось на його сусідніх "братів".
    - Випадок 4a: Запозичення (Redistribution). Якщо один із сусідніх братів є "багатим" (має більше `t-1` ключів), ми "запозичуємо" у нього один ключ. Цей ключ переміщується не напряму, а через

батьківський вузол за допомогою обертання, щоб зберегти порядок.

- Випадок 4б: Злиття (Merge). Якщо обидва сусіди також “бідні” (мають рівно `t-1` ключів), ми зливаємо наш вузол `L` з одним із сусідів. При цьому ключ-розділювач з батьківського вузла “опускається” вниз і стає середнім ключем нового об’єднаного вузла. Ця операція може спричинити недоповнення у батьківському вузлі, і процес відновлення може рекурсивно поширитися вгору аж до кореня.

Псевдокод:

```
ФУНКЦІЯ Delete(node, key):
    // 1. Знайти ключ 'key' у вузлі 'node' або визначити піддерево дл
я спуску.

    // 2. Якщо ключ у поточному вузлі 'node':
    ЯКЩО key в node.keys:
        ЯКЩО node.is_leaf: // Сценарій 1
            Видалити_ключ_з_листка(node, key)
        ІНАКШЕ: // Сценарій 2
            // Знайти наступника, поміняти місцями, рекурсивно видали
ти з піддерева
            // ...
        КІНЕЦЬ ЯКЩО
    // 3. Якщо ключ не тут, рекурсивно спускаємось
    ІНАКШЕ:
        child_node = відповідний_нащадок(node, key)
        // Проактивне виправлення: перед спуском гарантуємо, що нащад
ок має t ключів
        ЯКЩО child_node.num_keys < t:
            fix_underflow_proactively(node, child_index)
        КІНЕЦЬ ЯКЩО
        Delete(оновлений_нащадок, key)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Примітка: Існує два підходи до видалення — з реактивним виправленням “знизу-вгору” або з проактивним “зверху-вниз”. Другий підхід є складнішим, але вимагає лише одного проходу по дереву.

Обхід B дерева

Симетричний обхід

Логіка симетричного обходу.

Алгоритм є узагальненням симетричного обходу для бінарного дерева. Для вузла, що містить `k` ключів та `k+1` нащадків, логіка така:

1. Рекурсивно обійти перше піддерево (`child_0`).
2. Відвідати перший ключ (`key_0`).
3. Рекурсивно обійти друге піддерево (`child_1`).
4. Відвідати другий ключ (`key_1`).
5. ...і так далі, доки не будуть відвідані всі ключі та піддерева.

Псевдокод

```
// Вхід: node - вузол для обходу.
ФУНКЦІЯ InOrderTraversal(node):
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    i = 0
    // Проходимо по всіх ключах і нащадках
    ПОКИ i < node.num_keys:
        // 1. Рекурсивно обходимо піддерево, що знаходиться зліва від
ключа
        ЯКЩО node.is_leaf == false ТО
            InOrderTraversal(node.children[i])
        КІНЕЦЬ ЯКЩО

        // 2. Відвідуємо сам ключ
        Обробити(node.keys[i])

        i = i + 1
    КІНЕЦЬ ПОКИ

    // 3. Обходимо останнє, найправіше піддерево
    ЯКЩО node.is_leaf == false ТО
        InOrderTraversal(node.children[i])
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Прямий обхід

Цей обхід слідує логіці “спочатку батько, потім нащадки”. Для B-дерева це означає, що ми спочатку обробляємо всі ключі у поточному вузлі, і лише після цього рекурсивно спускаємось до його нащадків.

Застосування: Цей порядок є корисним для створення точної копії дерева, оскільки він дозволяє вставляти вузли в нове дерево в тому ж порядку, в якому вони були в оригіналі, що може відновити його структуру.

Псевдокод

```
// Вхід: node - вузол для обходу.
ФУНКЦІЯ PreOrderTraversal(node):
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // 1. Спочатку відвідуємо ВСІ ключі в поточному вузлі.
    ДЛЯ i ВІД 0 ДО node.num_keys-1:
        Обробити(node.keys[i])
    КІНЕЦЬ ДЛЯ

    // 2. Потім рекурсивно обходимо ВСІХ нащадків.
    ЯКЩО node.is_leaf == false ТО
        ДЛЯ i ВІД 0 ДО node.num_keys:
            PreOrderTraversal(node.children[i])
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Зворотний обхід

Цей обхід слідує логіці “спочатку нащадки, потім батько”. Для B-дерева це означає, що ми спочатку рекурсивно обходимо всі піддерева, і лише після того, як ми повернулись, обробляємо ключі поточного вузла.

Застосування: Основне застосування цього обходу — безпечне видалення всього дерева. Ми не можемо видалити батьківський вузол, доки не видалили всіх його нащадків. Зворотний обхід гарантує, що ми обробимо (і зможемо видалити) вузол лише після того, як всі його дочірні піддерева були оброблені.

Псевдокод

```
// Вхід: node - вузол для обходу.
ФУНКЦІЯ PostOrderTraversal(node):
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // 1. Спочатку рекурсивно обходимо ВСІХ нащадків.
    ЯКЩО node.is_leaf == false ТО
```

```
        ДЛЯ i ВІД 0 ДО node.num_keys:
            PostOrderTraversal(node.children[i])
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ЯКЩО

    // 2. Лише після цього відвідуємо ВСІ ключі в поточному вузлі.
    ДЛЯ i ВІД 0 ДО node.num_keys-1:
        Обробити(node.keys[i])
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності операцій у класичному B-дереві

Фундаментальна перевага B-дерева полягає в тому, що воно спроєктоване з однією головною метою: мінімізувати кількість повільних операцій доступу до диска. Весь аналіз його продуктивності базується на цьому принципі та на його гарантовано низькій висоті.

Фундамент аналізу: Висота B-дерева

Як і для інших збалансованих дерев, ефективність операцій у B-дереві напряму залежить від його висоти (h). Однак, на відміну від бінарних дерев, B-дерево є значно “нижчим” та “ширшим”.

Давайте проаналізуємо, чому. У B-дереві порядку `t` кожен внутрішній вузол (окрім кореня) має щонайменше `t` нащадків. Це означає, що на кожному наступному рівні кількість вузлів зростає щонайменше в `t` разів. Для дерева з `N` елементів, його висота h буде обмежена величиною:

$$h \leq \log_t \frac{N+1}{2}$$

З точки зору асимптотичного аналізу, основа логарифма не має значення, оскільки логарифми з різними основами відрізняються лише на константний множник. Тому для всіх B-дерев ми можемо стверджувати, що їхня висота $h = O(\log N)$.

Однак на практиці основа логарифма `t` є надзвичайно важливою. Якщо порядок `t = 500`, то висота дерева буде в сотні разів меншою, ніж у бінарного дерева. Це означає в сотні разів менше дискових доступів, що і є головною метою.

Пошук

– Найгірший випадок: $O(\log_t N)$ Алгоритм пошуку завжди проходить один шлях від кореня до вузла, в якому знаходиться ключ, або до листка. Довжина цього шляху дорівнює висоті дерева h (або `h+1`). На

кожному рівні ми зчитуємо з диска один вузол (сторінку) і виконуємо пошук всередині нього. Пошук у вузлі, де знаходиться до `2t-1` ключів, можна виконати за $O(\log t)$ (бінарний пошук) або $O(t)$ (лінійний пошук). Оскільки t є константою, що не залежить від N, вартість обробки одного вузла є константною. Таким чином, загальна складність визначається кількістю дискових доступів, тобто висотою дерева.

– Середній випадок: $O(\log_t N)$ Оскільки B-дерево завжди збалансоване, середня довжина шляху також є логарифмічною.

Вставка

– Найгірший випадок: $O(\log_t N)$ Процес вставки в B-дерево з проактивним розщепленням складається з таких етапів:

1. Спуск по дереву: Ми проходимо один шлях від кореня до листка. Це займає $O(h) = O(\log_t N)$ кроків (дискових читань).
2. Розщеплення на шляху: На кожному кроці ми можемо натрапити на переповнений вузол і виконати його розщеплення. Операція `splitChild` є локальною і вимагає константної кількості операцій з масивами та вказівниками, що займає $O(t)$ часу в пам'яті, та трьох дискових записів (для батька та двох нових нащадків).
3. Вставка в листок: Вставка ключа у масив ключів листка також займає $O(t)$.

Оскільки операції всередині вузлів (`splitChild`, пошук) мають складність, залежну від `t` (константа), а не від `N`, загальна складність визначається кількістю відвіданих вузлів, тобто висотою. Таким чином, складність вставки становить $O(\log_t N)$.

– Середній випадок (амортизований): $O(\log_t N)$ Розщеплення вузлів відбуваються відносно рідко. Більшість вставок не викликають жодних розщеплень. Тому амортизована вартість вставки також є логарифмічною.

Видалення

– Найгірший випадок: $O(\log_t N)$ Логіка аналізу схожа на вставку:

1. Пошук елемента: Перший крок — знайти вузол, що містить ключ для видалення. Це займає $O(\log_t N)$.

2. Заміна (за потреби): Якщо ключ знаходиться не в листку, ми міняємо його місцями з його наступником, який гарантовано знаходиться в листку. Пошук наступника також вимагає спуску по дереву, що займає $O(\log_t N)$.
3. Балансування: У найгіршому випадку, видалення може спричинити недоповнення вузла. Це вимагає або запозичення ключа у сусіда, або злиття з ним. Кожна така операція є локальною ($O(t)$) і може викликати недоповнення у батьківському вузлі. Цей процес може поширитися вгору аж до кореня, вимагаючи до $O(h) = O(\log_t N)$ операцій балансування.

Таким чином, сумарна складність видалення також становить $O(\log_t N)$.

– Середній випадок (амортизований): $O(\log_t N)$ Аналогічно до вставки, операції злиття та перерозподілу відбуваються нечасто, тому середня вартість залишається логарифмічною.

Обходи дерева

– Часова складність: $O(N)$ Для будь-якого повного обходу дерева нам потрібно відвідати кожен ключ у дереві рівно один раз. Якщо в дереві N ключів, то і складність буде лінійною відносно N. Це вимагатиме зчитування всіх вузлів дерева з диска.

– Просторова складність:

- DFS (рекурсивні обходи): Визначається максимальною глибиною стека рекурсії, яка дорівнює висоті дерева. Для B-дерева це $O(\log_t N)$.
- BFS (обхід в ширину): Визначається максимальним розміром черги, що може сягати кількості вузлів на останньому рівні. У найгіршому випадку просторова складність — $O(N)$.

Операції у B+-дереві

Логіка операцій у B+-дереві базується на тих самих принципах, що й у класичному B-дереві (пошук, розщеплення, злиття), але має важливі модифікації, пов’язані з двома його ключовими властивостями:

1. Всі дані (або вказівники на них) зберігаються виключно у листках.
2. Всі листки зв’язані у двозв’язний список.

Операція Пошуку

Пошук у B+-дереві завжди є шляхом від кореня до одного з листкових вузлів, оскільки лише там знаходиться потрібна нам інформація.

Процедура:

1. Починаємо з кореня.
2. На кожному внутрішньому вузлі ми виконуємо пошук серед його ключів, щоб знайти правильний вказівник на нащадка. Якщо наш шуканий ключ `key` менший за перший ключ вузла, ми йдемо за першим вказівником; якщо він між ключами K_i та K_{i+1}, ми йдемо за відповідним вказівником між ними, і так далі.
3. Ми повторюємо крок 2, спускаючись по дереву, доки не досягнемо листкового вузла.
4. У листковому вузлі ми виконуємо лінійний (або бінарний) пошук серед його ключів, щоб знайти точний збіг.

Псевдокод:

```
// Вхід: node - поточний вузол, key - ключ для пошуку.
// Вихід: вказівник на дані, пов'язані з ключем, або NULL.
ФУНКЦІЯ Search(node, key):
    // Поки поточний вузол не є листком, спускаємось по дереву.
    ПОКИ node.is_leaf == false:
        // Знаходимо правильний індекс нащадка для переходу.
        i = 0
        ПОКИ i < node.num_keys І key >= node.keys[i]:
            i = i + 1
        КІНЕЦЬ ПОКИ
        // Переходимо до нащадка.
        node = node.children[i]
    КІНЕЦЬ ПОКИ

    // Тепер ми в листковому вузлі. Шукаємо ключ тут.
    ДЛЯ і ВІД 0 ДО node.num_keys-1:
        ЯКЩО node.keys[i] == key ТО
            ПОВЕРНУТИ node.data_pointers[i] // Знайдено!
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    ПОВЕРНУТИ NULL // Не знайдено
КІНЕЦЬ ФУНКЦІЇ
```

Операція Вставки

Вставка у B+-дереві завжди відбувається в листковий вузол.

Процедура:

1. Пошук місця: Спочатку ми виконуємо пошук, як описано вище, щоб знайти правильний листковий вузол `L`, куди потрібно вставити новий ключ.
2. Вставка в листок: Вставляємо новий ключ (та пов'язані з ним дані) у листок `L`, зберігаючи відсортований порядок ключів.
3. Обробка переповнення:
    - Якщо у листку `L` є вільне місце, операція завершена.
    - Якщо листок `L` переповнений (тепер у ньому `2t` ключів), ми маємо його розщепити:
        a. Створюється новий листковий вузол `L'`.
        b. Перші `t` ключів залишаються в `L`, решта `t` ключів переміщуються в `L'`.
        c. Копія середнього ключа (тепер першого ключа в `L'`) "виштовхується" вгору до батьківського вузла.
        d. Новий вузол `L'` вставляється у двозв'язний список після `L`.
        e. Вставка ключа-розділювача в батьківський вузол може, у свою чергу, спричинити його переповнення та розщеплення, і цей процес може рекурсивно поширюватися вгору аж до кореня. Якщо розщеплюється корінь, створюється новий корінь, і висота дерева збільшується на 1.

Псевдокод:

```
ФУНКЦІЯ Insert(root, key, data):
    // Якщо корінь переповнений, розщеплюємо його перед спуском
    ЯКЩО root.num_keys == 2*t - 1 ТО
        new_root = Створити_Вузол(is_leaf=false)
        new_root.children[0] = root
        splitChild(new_root, 0)
        root = new_root
    КІНЕЦЬ ЯКЩО
    insertNonFull(root, key, data)
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ insertNonFull(node, key, data):
    // Якщо поточний вузол є листком
    ЯКЩО node.is_leaf == true ТО
```

```
        // Вставляємо ключ і дані у відсортований масив ключів листка
        // ... логіка вставки в масив ...
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // Інакше, знаходимо нащадка для спуску
    і = знайти_індекс_піддерева(node, key)

    // Якщо нащадок переповнений, розщеплюємо його
    ЯКЩО node.children[i].num_keys == 2*t - 1 ТО
        splitChild(node, i)
        // Після розщеплення потрібно перевірити, в яку з двох нових
гілок йти
        ЯКЩО key > node.keys[i] ТО
            і = і + 1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ЯКЩО

    // Рекурсивно викликаємо для відповідного нащадка
    insertNonFull(node.children[i], key, data)
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ splitChild(parent, child_index):
    // child - переповнений вузол, який треба розщепити
    child = parent.children[child_index]

    // Створюємо нового "брата"
    new_sibling = Створити_Вузол(is_leaf=child.is_leaf)
    new_sibling.num_keys = t - 1

    // Копіюємо останню половину ключів зі старого вузла в новий
    ДЛЯ j ВІД 0 ДО t-2:
        new_sibling.keys[j] = child.keys[j + t]
    КІНЕЦЬ ДЛЯ

    // Якщо це не листок, копіюємо також вказівники на нащадків
    ЯКЩО child.is_leaf == false ТО
        ДЛЯ j ВІД 0 ДО t-1:
            new_sibling.children[j] = child.children[j + t]
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ЯКЩО

    // Оновлюємо кількість ключів у старому вузлі
    child.num_keys = t - 1
```

```
    // Зсуваємо вказівники на нащадків у батьківському вузлі, щоб зві
льнити місце
    ДЛЯ j ВІД parent.num_keys ДО child_index + 1 кроком -1:
        parent.children[j + 1] = parent.children[j]
    КІНЕЦЬ ДЛЯ

    // Вставляємо вказівник на нового брата
    parent.children[child_index + 1] = new_sibling

    // Зсуваємо ключі у батьківському вузлі
    ДЛЯ j ВІД parent.num_keys-1 ДО child_index кроком -1:
        parent.keys[j + 1] = parent.keys[j]
    КІНЕЦЬ ДЛЯ

    // "Піднімаємо" середній ключ у батьківський вузол
    parent.keys[child_index] = child.keys[t - 1]
    parent.num_keys = parent.num_keys + 1
КІНЕЦЬ ФУНКЦІЇ
Операція Видалення
```

Видалення, як і вставка, завжди починається з листкового вузла.

Процедура:

1. Пошук: Знаходимо листковий вузол `L`, що містить ключ `key`, який потрібно видалити.
2. Видалення з листка: Видаляємо ключ та пов’язані з ним дані з `L`.
3. Обробка недоповнення:
    - Якщо листок `L` все ще містить щонайменше `t-1` ключів, операція завершена. Можливо, доведеться оновити ключ-розділювач у батьківському вузлі, якщо ми видалили найменший елемент у `L`.
    - Якщо листок `L` став недоповненим (має менше `t-1` ключів), ми намагаємося відновити баланс:
        a. Запозичення (Redistribution): Спочатку ми намагаємося “запозичити” ключ у сусіднього “багатого” братнього вузла (в якому більше `t-1` ключів). Ми переміщуємо один ключ від брата до нашого вузла через батьківський вузол.
        b. Злиття (Merge): Якщо обидва сусіди також “бідні” (мають рівно `t-1` ключів), ми зливаємо наш вузол з одним із сусідів. При цьому з батьківського вузла видаляється ключ-розділювач, що стояв між ними.

c. Цей процес (видалення ключа з батька) може, у свою чергу, спричинити недоповнення батьківського вузла, і процес балансування може поширюватися вгору аж до кореня. Якщо корінь залишається з одним нащадком, він видаляється, і висота дерева зменшується.

Псевдокод:

```
ФУНКЦІЯ Delete(root, key):
    // ... логіка знаходження та видалення ключа з листка L ...
    // Ця частина також має проактивно заповнювати вузли на шляху вни
з,
    // щоб гарантувати, що видалення з листка не призведе до недоповн
ення.

    // Альтернативний підхід "знизу-вгору":
    // 1. Знайти і видалити ключ з листка L.
    // 2. Якщо виникло недоповнення, викликати функцію виправлення.
    ЯКЩО L.num_keys < t - 1 ТО
        handleUnderflow(L)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ handleUnderflow(node):
    // ... логіка, що перевіряє сусідніх братів ...
    // ЯКЩО сусід "багатий" ТО
    //     Redistribute(...) // Запозичити
    // ІНАКШЕ
    //     Merge(...) // Злиття
    //     // Рекурсивно викликати handleUnderflow для батька, якщо п
отрібно
    // КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Операція Діапазонного запиту

Це операція, де B+-дерево демонструє свою головну перевагу.

Процедура:

1. Пошук початку: За допомогою звичайного пошуку ($O(\log_t N)$) знаходимо листковий вузол, що містить початковий ключ нашого діапазону.
2. Послідовний обхід: Починаючи з цього вузла, ми просто рухаємось вправо по двозв’язному списку листків, зчитуючи всі дані послідовно, доки не вийдемо за межі діапазону.

Псевдокод:

```
ФУНКЦІЯ RangeScan(root, start_key, end_key):
    // 1. Знаходимо початковий листок
    start_leaf = знайти_листок_для(root, start_key)

    results = []
    current_leaf = start_leaf

    // 2. Ітеруємо по зв'язаному списку листків
    ПОКИ current_leaf != NULL:
        // Ітеруємо по ключах всередині поточного листка
        ДЛЯ i ВІД 0 ДО current_leaf.num_keys-1:
            key = current_leaf.keys[i]
            // Додаємо до результату, якщо ключ в діапазоні
            ЯКЩО key >= start_key І key <= end_key ТО
                Додати(results, current_leaf.data_pointers[i])
            // Якщо вийшли за межі діапазону, завершуємо
            ІНАКШЕ ЯКЩО key > end_key ТО
                ПОВЕРНУТИ results
            КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ДЛЯ
        // Переходимо до наступного листка
        current_leaf = current_leaf.next_leaf
    КІНЕЦЬ ПОКИ

    ПОВЕРНУТИ results
КІНЕЦЬ ФУНКЦІЇ
```

Ця операція є надзвичайно ефективною, оскільки після початкового пошуку решта роботи — це швидке послідовне читання з диска, яке є набагато ефективнішим за випадкові доступи.

Обхід B+-дерева

Обходи для B+-дерева мають свою специфіку, пов’язану з його унікальною структурою, де всі дані знаходяться лише в листках.

Симетричний обхід

Для B+-дерева симетричний обхід, який повертає дані у відсортованому порядку, реалізується набагато ефективніше, ніж у класичному B-дереві. Завдяки зв’язаному списку листків, нам не потрібно рекурсивно обходити все дерево.

Ефективний алгоритм:

1. Знайти найлівіший (перший) листковий вузол. Це робиться одним спуском від кореня.
2. Пройтись по зв’язаному списку листків до самого кінця, обробляючи всі ключі та дані в кожному вузлі.

Цей підхід є значно швидшим, оскільки він уникає складних рекурсивних викликів та підйомів/спусків по дереву. Це, по суті, лінійний прохід по відсортованих даних.

Псевдокод такий:

```
// Вхід: root - корінь дерева.
ФУНКЦІЯ InOrderTraversal_BPlus(root):
    // 1. Знаходимо найлівіший листок
    node = root
    ПОКИ node.is_leaf == false:
        node = node.children[0]
    КІНЕЦЬ ПОКИ

    // 2. Ітеруємо по зв'язаному списку листків до кінця
    ПОКИ node != NULL:
        // Обробляємо всі ключі в поточному листку
        ДЛЯ i ВІД 0 ДО node.num_keys-1:
            Обробити(node.keys[i], node.data_pointers[i])
        КІНЕЦЬ ДЛЯ

        // Переходимо до наступного листка
        node = node.next_leaf
    КІНЕЦЬ ПОКИ
КІНЕЦЬ ФУНКЦІЇ
```

Прямий та Зворотний обходи

Ці обходи мають обмежене практичне застосування для B+-дерев, оскільки вони відвідують вузли у порядку, який не є корисним для отримання відсортованих даних. Їх зазвичай використовують для системних завдань, як-от копіювання структури дерева або його видалення.

Логіка їх реалізації ідентична до класичного B-дерева, але важливо пам’ятати, що обробка самих даних відбувається лише на рівні листків.

Прямий обхід

Слідує логіці “спочатку батько, потім нащадки”.

```
ФУНКЦІЯ PreOrderTraversal_BPlus(node):
    ЯКЩО node == NULL ТО ПОВЕРНУТИ КІНЕЦЬ ЯКЩО
```

```
    // 1. Відвідуємо ключі поточного вузла
    // Цей крок обробляє ключі-розділювачі у внутрішніх вузлах
    // та ключі з даними у листках.
    ДЛЯ i ВІД 0 ДО node.num_keys-1:
        Обробити_ключ(node.keys[i])
        ЯКЩО node.is_leaf == true ТО
            Обробити_дані(node.data_pointers[i])
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    // 2. Рекурсивно обходимо нащадків
    ЯКЩО node.is_leaf == false ТО
        ДЛЯ i ВІД 0 ДО node.num_keys:
            PreOrderTraversal_BPlus(node.children[i])
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Зворотний обхід

Слідує логіці “спочатку нащадки, потім батько”. Ідеально підходить для видалення всіх вузлів дерева.

```
ФУНКЦІЯ PostOrderTraversal_BPlus(node):
    ЯКЩО node == NULL ТО ПОВЕРНУТИ КІНЕЦЬ ЯКЩО

    // 1. Рекурсивно обходимо нащадків
    ЯКЩО node.is_leaf == false ТО
        ДЛЯ i ВІД 0 ДО node.num_keys:
            PostOrderTraversal_BPlus(node.children[i])
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ЯКЩО

    // 2. Відвідуємо ключі та дані поточного вузла
    ДЛЯ i ВІД 0 ДО node.num_keys-1:
        Обробити_ключ(node.keys[i])
        ЯКЩО node.is_leaf == true ТО
            Обробити_дані(node.data_pointers[i])
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    // На цьому етапі вузол можна безпечно видалити
    // ВИДАЛИТИ(node)
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності операцій у B+-дереві

Фундаментальна перевага B+-дерева полягає в тому, що воно спроєктоване з однією головною метою: мінімізувати кількість повільних операцій доступу до диска. Весь аналіз його продуктивності базується на цьому принципі та на його гарантовано низькій висоті.

Фундамент аналізу: Висота B+-дерева

Як і для класичного B-дерева, висота h B+-дерева з N елементів та порядком t є логарифмічною. Оскільки кожен вузол (окрім кореня) має від t до $2t$ нащадків, кількість вузлів на кожному рівні зростає експоненційно. Це призводить до того, що висота дерева h є $O(\log_t N)$.

Чому основа логарифма t важлива? У контексті Big O основа логарифма зазвичай не має значення, оскільки $\log_t N = \frac{\log_2 N}{\log_2 t}$, а $\log_2 t$ є константою. Однак при аналізі дискових структур основа має величезне практичне значення. Якщо порядок `t = 500`, то висота дерева буде в $\log_2 500 \approx 9$ разів меншою, ніж у бінарного дерева. Це означає в 9 разів менше дискових доступів! Тому, хоча асимптотично складність $O(\log N)$, на практиці ми завжди пам'ятаємо про цей великий дільник.

Пошук

- Найгірший випадок: $O(\log_t N)$ Алгоритм пошуку завжди проходить один шлях від кореня до листка. Довжина цього шляху дорівнює висоті дерева h. На кожному рівні ми зчитуємо з диска один вузол (сторінку) і виконуємо пошук всередині нього. Пошук у вузлі, де знаходиться до `2t-1` ключів, можна виконати за $O(\log t)$ (бінарний пошук) або $O(t)$ (лінійний пошук). Оскільки t є константою, що не залежить від N, вартість обробки одного вузла є константною. Таким чином, загальна складність визначається кількістю дискових доступів, тобто висотою дерева.
- Середній випадок: $O(\log_t N)$ Оскільки B+-дерево завжди збалансоване, середня довжина шляху також є логарифмічною.

Вставка

- Найгірший випадок: $O(\log_t N)$ Процес вставки складається з декількох етапів:
    1. Пошук листка: Спочатку потрібно знайти правильний листковий вузол для вставки. Це вимагає проходу від кореня до листка, що коштує $O(\log_t N)$ дискових доступів.

2. Вставка в листок: Сама вставка в масив ключів листка є операцією в пам'яті, що займає $O(t)$.
3. Розщеплення та підйом: У найгіршому випадку, вставка може спричинити переповнення листка. Це призводить до його розщеплення ($O(t)$) та вставки нового ключа-розділювача у батьківський вузол. Ця вставка, у свою чергу, може викликати розщеплення батька, і цей процес може поширитися вгору аж до самого кореня. Оскільки кількість рівнів у дереві — $O(\log_t N)$, у найгіршому випадку нам доведеться виконати $O(\log_t N)$ операцій розщеплення.

Загальна складність є сумою цих етапів: $O(\log_t N)$ на пошук та $O(t \cdot \log_t N)$ на розщеплення. Оскільки t є константою, фінальна складність становить $O(\log_t N)$.

– Середній випадок (амортизований): $O(\log_t N)$ Розщеплення вузлів відбуваються відносно рідко. Більшість вставок не викликають жодних розщеплень. Тому амортизована вартість вставки також є логарифмічною.

Видалення

– Найгірший випадок: $O(\log_t N)$ Логіка аналізу схожа на вставку:
1. Пошук листка: $O(\log_t N)$ дискових доступів.
2. Видалення з листка: $O(t)$ операцій в пам'яті.
3. Балансування (злиття або перерозподіл): У найгіршому випадку видалення може спричинити недоповнення листка. Це вимагає або запозичення ключа у сусіда, або злиття з ним ($O(t)$). Ця операція може викликати недоповнення у батьківському вузлі, і процес балансування може поширитися вгору аж до кореня. Максимальна кількість операцій злиття/перерозподілу дорівнює висоті дерева.

Таким чином, загальна складність видалення також становить $O(\log_t N)$.

– Середній випадок (амортизований): $O(\log_t N)$ Аналогічно до вставки, операції злиття та перерозподілу відбуваються нечасто, тому середня вартість залишається логарифмічною.

Діапазонний запит

Це операція, де B+-дерево демонструє свою ключову перевагу.

Найгірший випадок: $O(\log_t N + K)$ Де N — загальна кількість елементів у дереві, а K — кількість елементів у діапазоні, що повертається.

1. Пошук початку діапазону: Спочатку ми виконуємо стандартний пошук, щоб знайти перший листок, який містить початок нашого діапазону. Це займає $O(\log_t N)$ часу.

2. Послідовне сканування: Після знаходження першого елемента, ми просто рухаємось по зв'язаному списку листків. Це швидке послідовне читання з диска. Якщо нам потрібно повернути K елементів, ми відвідаємо приблизно K/t листкових вузлів. Тому ця частина має складність $O(K)$ (з точки зору кількості елементів) або $O(K/t)$ (з точки зору дискових доступів).

Загальна складність є сумою цих двох фаз. Це надзвичайно ефективно, оскільки ми не виконуємо повторних логарифмічних пошуків для кожного елемента в діапазоні.

- Середній випадок: Аналогічний до найгіршого.

Просторова складність Просторова складність B+-дерева визначається кількістю елементів, які воно зберігає. Для N елементів вона становить $O(N)$. Самі операції (пошук, вставка, видалення) є рекурсивними або ітеративними, і їхня додаткова просторова складність пропорційна глибині рекурсії (висоті дерева), тобто $O(\log_t N)$.

Операції у B*-дереві

Операції у B*-дереві базуються на тих самих принципах, що й у класичному B-дереві, але з однією ключовою відмінністю: алгоритм докладає максимум зусиль, щоб уникнути розщеплення або злиття вузлів, віддаючи перевагу перерозподілу ключів між сусідніми вузлами. Це дозволяє підтримувати інваріант, згідно з яким кожен вузол (окрім кореня) має бути заповнений щонайменше на 2/3.

Операція Пошуку

Алгоритм пошуку в B*-дереві є абсолютно ідентичним до алгоритму пошуку в класичному B-дереві. Оскільки B*-дерево є лише його структурною оптимізацією, властивість бінарного дерева пошуку для ключів у вузлах зберігається.

Процедура:

1. Починаємо з кореня і спускаємось по дереву.

2. На кожному внутрішньому вузлі шукаємо ключ, щоб визначити, в яке з піддерев потрібно перейти.
3. Процес триває, доки ми не знайдемо шуканий ключ у внутрішньому або листковому вузлі, або доки не дійдемо до листка, де ключа немає, що означає його відсутність у дереві.

Псевдокод:

```
// Псевдокод ідентичний до пошуку в класичному B-дереві.
// Вхід: node - поточний вузол, key - ключ для пошуку.
// Вихід: пара (вузол, індекс), де знайдено ключ, або NULL.
ФУНКЦІЯ Search(node, key):
    i = 0
    ПОКИ i < node.num_keys І key > node.keys[i]:
        i = i + 1
    КІНЕЦЬ ПОКИ

    ЯКЩО i < node.num_keys І key == node.keys[i] ТО
        ПОВЕРНУТИ (node, i) // Знайдено
    КІНЕЦЬ ЯКЩО

    ЯКЩО node.is_leaf == true ТО
        ПОВЕРНУТИ NULL // Не знайдено
    КІНЕЦЬ ЯКЩО

    child_node = DISK_READ(node.children[i])
    ПОВЕРНУТИ Search(child_node, key)
КІНЕЦЬ ФУНКЦІЇ
```

Операція Вставки

Вставка — це операція, де проявляється головна відмінність B*-дерева. Алгоритм намагається відкласти розщеплення до останнього можливого моменту.

Процедура:

1. Пошук місця: Виконується стандартний пошук, щоб знайти листковий вузол L, куди потрібно вставити новий ключ.
2. Вставка в листок: Новий ключ вставляється в масив ключів вузла L.
3. Обробка переповнення:
    - Якщо у листку L є вільне місце, операція завершена.

- Якщо листок `L` переповнився (містить `2t-1` ключів після вставки), ми дивимось на його сусідніх "братів" (лівого або правого).
    - Крок 3a: Перерозподіл (Redistribution). Якщо сусідній брат має вільне місце (містить менше, ніж `2t-2` ключів), відбувається перерозподіл. Частина ключів з переповненого вузла `L` переміщується до брата через їхнього спільного батька. Це локальне перебалансування, яке не змінює висоту дерева.
    - Крок 3b: Розщеплення 2 на 3. Якщо сусідній брат також повний, перерозподіл неможливий. Лише в цьому випадку відбувається розщеплення. Але це не звичайне розщеплення "1 на 2". Натомість, два повні вузли (наш `L` та його брат) об'єднуються і розщеплюються на три нові вузли, кожен з яких буде заповнений приблизно на 2/3. Два ключі-розділювачі піднімаються до батьківського вузла.
4. Поширення вгору: Вставка нових ключів у батьківський вузол може, у свою чергу, спричинити його переповнення, і вся логіка (перерозподіл або розщеплення 2 на 3) рекурсивно застосовується на шляху вгору до кореня.

Псевдокод:

```
ФУНКЦІЯ Insert(root, key):
    // ... логіка знаходження листка L для вставки ...

    Вставити_ключ_в_листок(L, key)

    ЯКЩО L.num_keys > 2*t - 2 ТО // Перевірка на переповнення
        handleOverflow(L)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ handleOverflow(node):
    parent = node.parent
    // Знаходимо сусіднього брата 'sibling'
    sibling = знайти_сусіда(node)

    // Крок 3a: Перерозподіл
    ЯКЩО sibling != NULL І sibling.num_keys < 2*t - 2 ТО
        Redistribute(node, sibling, parent)
```

```
    // Крок 3b: Розщеплення 2 на 3
    ІНАКШЕ
        splitTwoNodesToThree(node, sibling, parent)
        // Рекурсивно перевіряємо батька на переповнення
        ЯКЩО parent.num_keys > 2*t - 2 ТО
            handleOverflow(parent)
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Операція Видалення

Видалення в B*-дереві працює за дзеркальною логікою. Основна мета — підтримувати мінімальне заповнення на рівні 2/3, відкладаючи злиття вузлів до останнього.

Процедура:

1. Пошук та видалення: Виконується стандартна процедура видалення ключа (якщо ключ у внутрішньому вузлі, він замінюється наступником з листка, і задача зводиться до видалення з листка).
2. Обробка недоповнення:
    - Якщо вузол `N` після видалення все ще задовольняє інваріанту B*-дерева (має достатньо ключів), операція завершена.
    - Якщо вузол `N` став недоповненим, ми дивимось на його сусідніх братів.
        - Крок 2a: Запозичення (Redistribution). Якщо сусідній брат є “багатим” (має більше ключів, ніж мінімально необхідно), ми “запозичуємо” у нього ключ, переміщуючи його через батьківський вузол, щоб відновити баланс у нашому вузлі `N`.
        - Крок 2b: Злиття 3 на 2. Якщо сусідній брат також “бідний” (має мінімально дозволену кількість ключів), запозичення неможливе. Тоді ми виконуємо операцію, обернену до розщеплення: ми зливаємо три вузли (наш `N`, його “бідного” брата та їхнього спільного батька) у два нові вузли. Цей процес може рекурсивно поширюватися вгору по дереву.

Псевдокод:

```
ФУНКЦІЯ Delete(root, key):
    // ... логіка знаходження та видалення ключа з вузла N ...
```

```
    ЯКЩО N.num_keys < (2*t - 2) / 3 ТО // Перевірка на недоповнення
        handleUnderflow(N)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ

ФУНКЦІЯ handleUnderflow(node):
    parent = node.parent
    sibling = знайти_сусіда(node)

    // Крок 2a: Запозичення
    ЯКЩО sibling.num_keys > (2*t - 2) / 3 ТО
        Redistribute(node, sibling, parent)
    // Крок 2b: Злиття
    ІНАКШЕ
        mergeThreeNodesToTwo(node, sibling, parent)
        // Рекурсивно перевіряємо батька на недоповнення
        ЯКЩО parent.num_keys < (2*t - 2) / 3 ТО
            handleUnderflow(parent)
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Обхід B-дерева.*

Як і для B-дерева, єдиним осмисленим обходом, що зберігає порядок, є симетричний. Важливо зазначити, що B*-дерево є структурною оптимізацією класичного B-дерева, яка стосується правил заповнення та балансування (розщеплення/злиття). Однак фундаментальна структура — зберігання ключів у внутрішніх вузлах та організація вказівників на нащадків — залишається такою ж.

Тому алгоритми обходу для B*-дерева логічно ідентичні до алгоритмів для класичного B-дерева.

Симетричний обхід

Це основний спосіб обходу, який дозволяє отримати всі ключі дерева у відсортованому порядку.

Логіка: Для кожного вузла ми рекурсивно обходимо піддерево, потім відвідуємо ключ, потім обходимо наступне піддерево, і так далі.

Псевдокод

```
// Вхід: node - вузол для обходу.
ФУНКЦІЯ InOrderTraversal_BStar(node):
    ЯКЩО node == NULL ТО
```

```
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    i = 0
    ПОКИ i < node.num_keys:
        // 1. Рекурсивно обходимо піддерево, що знаходиться зліва від
ключа
        ЯКЩО node.is_leaf == false ТО
            InOrderTraversal_BStar(node.children[i])
        КІНЕЦЬ ЯКЩО

        // 2. Відвідуємо сам ключ
        Обробити(node.keys[i])

        i = i + 1
    КІНЕЦЬ ПОКИ

    // 3. Обходимо останнє, найправіше піддерево
    ЯКЩО node.is_leaf == false ТО
        InOrderTraversal_BStar(node.children[i])
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Прямий обхід

Цей обхід слідує логіці “спочатку вузол, потім нащадки” і корисний для створення копії дерева.

Логіка: Спочатку обробляються всі ключі в поточному вузлі, а потім рекурсивно обходяться всі його нащадки.

Псевдокод

```
// Вхід: node - вузол для обходу.
ФУНКЦІЯ PreOrderTraversal_BStar(node):
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // 1. Спочатку відвідуємо ВСІ ключі в поточному вузлі.
    ДЛЯ i ВІД 0 ДО node.num_keys-1:
        Обробити(node.keys[i])
    КІНЕЦЬ ДЛЯ

    // 2. Потім рекурсивно обходимо ВСІХ нащадків.
    ЯКЩО node.is_leaf == false ТО
        ДЛЯ i ВІД 0 ДО node.num_keys:
```

```
            PreOrderTraversal_BStar(node.children[i])
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

Зворотний обхід

Цей обхід слідує логіці “спочатку нащадки, потім батько” і є ідеальним для безпечного видалення всього дерева.

Логіка: Спочатку рекурсивно обходяться всі піддерева, і лише потім обробляються ключі в поточному вузлі.

Псевдокод

```
// Вхід: node - вузол для обходу.
ФУНКЦІЯ PostOrderTraversal_BStar(node):
    ЯКЩО node == NULL ТО
        ПОВЕРНУТИ
    КІНЕЦЬ ЯКЩО

    // 1. Спочатку рекурсивно обходимо ВСІХ нащадків.
    ЯКЩО node.is_leaf == false ТО
        ДЛЯ і ВІД 0 ДО node.num_keys:
            PostOrderTraversal_BStar(node.children[i])
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ЯКЩО

    // 2. Лише після цього відвідуємо ВСІ ключі в поточному вузлі.
    ДЛЯ і ВІД 0 ДО node.num_keys-1:
        Обробити(node.keys[i])
    КІНЕЦЬ ДЛЯ

    // На цьому етапі вузол можна безпечно видалити
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності операцій у B*-дереві

Фундаментальна ідея B*-дерева — досягти щільнішого пакування даних, вимагаючи, щоб кожен вузол був заповнений щонайменше на 2/3, а не на 1/2. Це досягається за рахунок більш “розумної” стратегії вставки та видалення, яка віддає перевагу перерозподілу ключів між сусідніми вузлами, а не їх розщепленню чи злиттю. Давайте проаналізуємо, як це впливає на складність операцій.

Фундамент аналізу: Висота B*-дерева

Оскільки B*-дерево є збалансованим, його висота h також є логарифмічною. Більше того, через вищі вимоги до мінімального заповнення, B*-дерево зазвичай є ще нижчим, ніж класичне B-дерево того ж порядку `t` для однакової кількості елементів `N`.

Кількість нащадків у внутрішньому вузлі (окрім кореня) варіюється від $\lceil (2t-1)/3 \rceil + 1$ до `2t`. Таким чином, основа логарифма в оцінці висоти є навіть більшою, ніж у B-дереві, але асимптотично висота все одно оцінюється як $h = O(\log_t N)$. Цей факт, як і раніше, є основою для аналізу всіх операцій.

Пошук

- Найгірший випадок: $O(\log_t N)$ Алгоритм пошуку в B*-дереві є абсолютно ідентичним до алгоритму в класичному B-дереві. Він складається з послідовності дискових читань (по одному на кожен рівень дерева) та пошуку всередині завантаженого вузла. Оскільки висота дерева логарифмічна, а пошук у вузлі займає $O(t)$ або $O(\log t)$ (що є константою), загальна складність визначається кількістю дискових доступів.
- Середній випадок: $O(\log_t N)$ Аналогічно до найгіршого випадку, оскільки дерево завжди збалансоване.

Вставка

- Найгірший випадок: $O(\log_t N)$ Процес вставки є складнішим, але його асимптотика залишається логарифмічною.
    1. Пошук листка: Спуск по дереву до відповідного листка для вставки. Цей етап вимагає $O(h) = O(\log_t N)$ дискових читань.
    2. Вставка в листок: Сама вставка в масив ключів листка є операцією в пам’яті, що займає $O(t)$.
    3. Балансування: Це ключова відмінність.
        - Перерозподіл: У більшості випадків, коли вузол переповнюється, алгоритм намагається виконати перерозподіл ключів із сусіднім вузлом. Це локальна операція, що вимагає читання сусіда та запису двох змінених вузлів та їхнього батька (константна кількість дискових доступів).
        - Розщеплення 2 на 3: Лише якщо сусідній вузол також повний, відбувається розщеплення. Ця операція також є

локальною і вимагає константної кількості дискових записів. Процес балансування може поширитися вгору до кореня, що в найгіршому випадку вимагатиме $O(h)$ операцій балансування.

Загальна складність є сумою цих етапів: $O(\log_t N)$ на пошук та $O(\log_t N)$ на потенційні операції балансування на шляху до кореня. Отже, фінальна складність становить $O(\log_t N)$.

– Середній випадок (амортизований): $O(\log_t N)$ Операції розщеплення в B*-дереві відбуваються ще рідше, ніж у B-дереві, оскільки перерозподіл є значно частішим сценарієм. Це робить амортизовану вартість вставки дуже близькою до вартості одного пошуку.

Видалення

– Найгірший випадок: $O(\log_t N)$ Логіка аналізу дзеркальна до вставки:

1. Пошук та видалення: Знаходимо та видаляємо ключ з листка (можливо, попередньо замінивши його наступником). Цей етап коштує $O(\log_t N)$.
2. Балансування: Якщо вузол стає недоповненим (менше 2/3 заповнення), алгоритм намагається відновити баланс.
   - Перерозподіл: Спочатку він намагається “запозичити” ключ у “багатого” сусіда. Це локальна операція з константною кількістю дискових доступів.
   - Злиття 3 на 2: Лише якщо сусід також має мінімально можливу кількість ключів, відбувається злиття трьох вузлів (наш, сусідній та їхній батько) у два нові. Цей процес також може поширюватися вгору до кореня, вимагаючи до $O(h) = O(\log_t N)$ операцій балансування.

Сумарна складність видалення також становить $O(\log_t N)$.

– Середній випадок (амортизований): $O(\log_t N)$ Аналогічно до вставки, злиття відбуваються рідко, тому середня вартість залишається логарифмічною.

Обходи дерева

– Часова складність: $O(N)$ Як і для будь-якого дерева, для повного обходу потрібно відвідати кожен з N ключів, що вимагає зчитування всіх вузлів дерева.

– Просторова складність:
    - DFS (рекурсивні обходи): Визначається висотою дерева, тобто $O(\log_t N)$.
    - BFS (обхід в ширину): Визначається максимальним розміром черги, що може сягати $O(N)$.

Переваги та недоліки

Переваги:

– Оптимізація для диска: Головна перевага. Зменшення висоти дерева кардинально зменшує кількість повільних дискових операцій.

– Гарантована логарифмічна складність: Як і інші збалансовані дерева, B-дерева гарантують $O(\log N)$ для всіх основних операцій.

– Ефективні діапазонні запити (для B+): Зв'язані листки дозволяють дуже швидко отримувати дані у певному діапазоні.

– Високий коефіцієнт заповнення: Вузли завжди заповнені щонайменше наполовину (або на 2/3 у B*), що ефективно використовує дисковий простір.

Недоліки:

– Складність реалізації: Реалізація B-дерева, особливо операції видалення, є значно складнішою, ніж для бінарних дерев.

– Надлишковість для RAM: Для даних, що повністю вміщуються в оперативну пам'ять, переваги B-дерева є не такими значними. Великі вузли та складніша логіка пошуку всередині вузла можуть зробити його повільнішим за червоно-чорне дерево.

– Додаткова пам'ять: Вузли вимагають більше пам'яті для зберігання масивів ключів та вказівників.

Напишемо, на підсумок, про відмінності між цими деревами.

B-дерево

Як ми вже знаємо, B-дерево — це збалансоване дерево, яке мінімізує кількість дискових доступів, зберігаючи в кожному вузлі велику кількість ключів.

Ключові характеристики:

- Зберігання даних: Кожен вузол містить як ключі, так і пов'язані з ними дані (або вказівники на ці дані).

- Структура: Внутрішні вузли містять ключі-розділювачі та вказівники на нащадків. Листкові вузли не мають нащадків.
- Заповнення: Кожен вузол (окрім кореня) заповнений щонайменше наполовину.

B-дерево є чудовою універсальною структурою, але воно не є оптимальним для всіх типів запитів.

B+-дерево: Оптимізація для баз даних

B+-дерево — це найпопулярніша варіація, яка стала де-факто стандартом для індексів у реляційних базах даних (наприклад, PostgreSQL, MySQL InnoDB). Воно вводить дві фундаментальні зміни порівняно з класичним B-деревом.

Відмінність №1: Місце зберігання даних

– У B-дереві дані зберігаються як у внутрішніх, так і в листкових вузлах.

– У B+-дереві всі дані (або вказівники на них) зберігаються виключно у листкових вузлах. Внутрішні вузли містять лише копії ключів, які слугують “дороговказами” для навігації до правильного листка.

Що це дає?

1. Більший фактор розгалуження: Оскільки внутрішні вузли не несуть “зайвого вантажу” у вигляді даних, в один дисковий блок (сторінку) можна вмістити значно більше ключів-розділювачів. Це збільшує кількість нащадків у кожного вузла і, як наслідок, робить дерево ще “нижчим” та “ширшим”, зменшуючи кількість дискових доступів при пошуку.

2. Стабільний час пошуку: Будь-який пошук завжди доходить до листкового рівня, що робить час пошуку більш передбачуваним.

Відмінність №2: Зв’язані листки

– У B-дереві листкові вузли ніяк не пов’язані між собою.

– У B+-дереві всі листкові вузли з’єднані в двозв’язний список.

Що це дає? Це ключова оптимізація для діапазонних запитів. Уявіть, що вам потрібно виконати запит `SELECT FROM users WHERE age BETWEEN 30 AND 40;`.

- У B-дереві вам доведеться виконати декілька окремих пошуків або складний обхід піддерев, що може вимагати багаторазового підйому та спуску по дереву.

- У B+-дереві ви просто знаходите перший листок, що містить ключ `30`, а потім швидко проходите по зв'язаному списку листків, доки не вийдете за межі діапазону (`>40`). Це надзвичайно ефективна операція, яка вимагає лише одного початкового пошуку і подальшого послідовного читання.

B*-дерево: Оптимізація використання простору

B*-дерево — це подальша модифікація B-дерева, головною метою якої є покращення заповненості вузлів і, як наслідок, більш ефективне використання дискового простору.

Відмінність №1: Мінімальне заповнення

– У B-дереві кожен вузол заповнений щонайменше на 1/2 (або 50%).

– У B*-дереві кожен вузол заповнений щонайменше на 2/3 (або ~67%).

Відмінність №2: Стратегія розщеплення

Цей вищий коефіцієнт заповнення досягається завдяки розумнішій стратегії обробки переповнених вузлів.

- У B-дереві, коли вузол переповнюється, він негайно розщеплюється на два вузли, кожен з яких буде заповнений приблизно наполовину.
- У B*-дереві, коли вузол переповнюється, він спершу намагається перерозподілити ключі з сусіднім "братнім" вузлом. Він "ділиться" ключами з сусідом, щоб уникнути розщеплення. І лише якщо сусідній вузол також повністю заповнений, виконується розщеплення. Але це не звичайне розщеплення: два повні вузли розщеплюються на три вузли, кожен з яких буде заповнений приблизно на 2/3.

Що це дає?

1. Краще використання дискового простору: Вищий коефіцієнт заповнення означає менше "порожнього місця" у файлах індексів.

2. Менша висота дерева: Оскільки вузли "щільніші", для зберігання тієї ж кількості ключів потрібно менше вузлів, що може призвести до ще меншої висоти дерева.

3. Менше операцій розщеплення: Операція розщеплення є дорогою. B*-дерево виконує її рідше, що може покращити продуктивність на системах з великою кількістю операцій запису.

Характеристика	B-дерево (Класичне)	B+-дерево	B*-дерево
Зберігання даних	У всіх вузлах	Лише у листках	У всіх вузлах
Структура листків	Не зв’язані	Зв’язані у список	Не зв’язані
Мін. заповнення вузла	1/2	1/2	2/3
Стратегія вставки	Розщеплення 1 на 2	Розщеплення 1 на 2	Перерозподіл, потім розщеплення 2 на 3
Основна перевага	Універсальність	Ефективні діапазонні запити	Краще використання простору
Типове застосування	Файлові системи	Індекси в СУБД	Файлові системи, деякі СУБД

Контрольні запитання

1. У чому полягає основна мета використання B-дерев замість збалансованих бінарних дерев?
2. Опишіть, що таке “порядок `t`” для B-дерева.
3. Які дві ключові відмінності B+-дерева від класичного B-дерева?
4. Поясніть, чому B+-дерево є особливо ефективним для діапазонних запитів.
5. Яку проблему вирішує B*-дерево порівняно з B-деревом?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація B-дерева та операції вставки з розщепленням.

2. Мета

Практичне засвоєння логіки роботи небінарних збалансованих дерев, що оптимізовані для роботи з блоковими пристроями. Реалізація механізму розщеплення вузлів.

3. Завдання

Створити програму, що реалізує B-дерево (з фіксованим, невеликим порядком `t`, наприклад, `t=3`) та операцію вставки.

4. Опис

Програма має складатися з класу `BTreeNode` (з масивами для ключів та нащадків) та основного класу `BTree`.

5. Порядок виконання

1. Створіть клас `BTreeNode`.
2. Реалізуйте `insert(key)`: Головна функція, що перевіряє корінь на переповнення і викликає рекурсивного помічника.
3. Реалізуйте `insertNonFull(node, key)`: Рекурсивна функція, що спускається по дереву.
4. Реалізуйте `splitChild(...)`: Функція, що розщеплює переповненого нащадка.
5. Реалізуйте `search(key)`: Для перевірки коректності вставки.
6. Протестуйте: Вставляйте послідовно числа і після кожної вставки, що викликає розщеплення, виводьте структуру дерева.

6. Результат

Додаток, що реалізує B-дерево та коректно підтримує його інваріанти при вставці.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми.
2. Продемонструвати роботу додатку, показавши сценарій, що призводить до розщеплення кореня.
3. Бути готовим відповісти на питання:
    - Пояснити, як працює ваша функція `splitChild`.
    - Чому висота B-дерева зростає лише при розщепленні кореня?
    - Яка гарантована часова складність вашої операції вставки?

Тести

1. Головна мета B-дерев — це:

a) Мінімізувати використання пам’яті.

b) Мінімізувати кількість операцій читання/запису з диска.

c) Спростити реалізацію порівняно з бінарними деревами.

d) Забезпечити найшвидший пошук для даних в RAM.

2. У B+-дереві вся корисна інформація (дані або вказівники на них) зберігається:

a) У кореневому вузлі.

b) Рівномірно по всіх вузлах.

c) Лише у внутрішніх вузлах.

d) Лише у листкових вузлах.

3. Якщо B-дерево має порядок `t=3`, скільки максимум ключів може бути в одному вузлі?

a) 2

b) 3

c) 5

d) 6

4. Вставка в B-дерево з проактивним розщепленням є операцією:

a) Що виконується за один прохід “зверху-вниз”.

b) Що вимагає двох проходів: “вниз”, а потім “вгору”.

c) Яка не потребує розщеплень.

d) Яка має експоненційну складність.

5. Яка структура даних є узагальненням 2-3-4 дерева?

a) AVL-дерево.

b) Червоно-чорне дерево.

c) B-дерево.

d) Декартове дерево.

Відповіді на прості тести: 1-b, 2-d, 3-c (бо $2t - 1 = 2 * 3 - 1 = 5$), 4-a, 5-c.

Тести підвищеної складності

1. У B+-дереві внутрішні вузли НЕ містять вказівників на дані (або самих даних), а лише ключі-розділювачі. Як це впливає на характеристики дерева порівняно з B-деревом того ж порядку?

a) Ніяк не впливає.

b) Це збільшує висоту дерева, оскільки потрібно більше вузлів.

c) Це дозволяє збільшити фактор розгалуження (в один блок вміщується більше ключів), що призводить до меншої висоти дерева.

d) Це робить діапазонні запити неможливими.

2. Яка операція є причиною того, що B*-дерево в середньому має вищий коефіцієнт заповнення, ніж B-дерево?

a) Воно використовує обертання замість розщеплень.

b) При переповненні воно спершу намагається перерозподілити ключі з сусіднім братом, і розщеплює вузли лише коли обидва брати заповнені.

c) Воно дозволяє вузлам бути заповненими лише на 100%.

d) Воно використовує більший порядок `t`.

3. У B-дереві порядку `t=5`, вузол видаляється з листка, в якому було `t-1=4` ключі. Якою буде наступна дія алгоритму?

a) Жодних дій, оскільки вузол все ще є коректним.

b) Потрібно негайно виконати злиття з братом.

c) Потрібно спробувати “запозичити” ключ у сусіднього брата.

d) Потрібно розщепити цей вузол.

4. Чому операція видалення в B-дереві зазвичай реалізується як “bottom-up” (з виправленням на шляху вгору), на відміну від “top-down” вставки?

a) Це простіше в реалізації.

b) “Top-down” видалення неможливе.

c) Тому що заздалегідь невідомо, чи призведе видалення до “недоповнення”. Ми не можемо “проактивно” зливати вузли, бо це може виявитись непотрібним і неефективним.

d) Це дозволяє уникнути обертань.

5. У B+-дереві для виконання запиту “знайти всі записи, де `key > 100`” найефективніша стратегія — це:

a) Виконати повний обхід дерева.

b) Виконати пошук ключа 100, а потім пройти по зв’язному списку листків вправо до кінця.

c) Багаторазово викликати операцію пошуку для 101, 102, і т.д.

d) Виконати пошук в глибину.

6. Якщо розмір дискового блоку — 4096 байт, розмір ключа — 8 байт, а розмір вказівника на нащадка — 8 байт, яким буде оптимальний порядок `t` для B-дерева?

a) 256

b) 255

c) 128

d) 512

7. У чому головний недолік B-дерев порівняно з хеш-таблицями для пошуку за унікальним ключем?

a) B-дерева мають гіршу асимптотичну складність у найгіршому випадку.

b) B-дерева вимагають, щоб дані були відсортовані.

c) Хеш-таблиця забезпечує середню складність $O(1)$, що швидше, ніж $O(\log_t N)$ у B-дерева, для даних в RAM.

d) B-дерева не підтримують видалення.

8. Зв’язок між червоно-чорним деревом та 2-3-4 деревом полягає в тому, що:

a) Будь-яке 2-3-4 дерево можна перетворити на червоно-чорне, і навпаки.

b) Вони мають однакову висоту для однакового набору даних.

c) Обидва використовують обертання як основний механізм балансування.

d) Червоно-чорне дерево є узагальненням 2-3-4 дерева.

Відповіді на тести підвищеної складності:

1. *c)* Оскільки ключі у внутрішніх вузлах не потребують місця для даних, в один дисковий блок (сторінку) можна "упакувати" більше ключів-розділювачів і, відповідно, більше вказівників на нащадків. Це збільшує `t` і зменшує висоту.

2. *b)* Замість того, щоб одразу розщеплювати повний вузол на два напівпорожні, B*-дерево намагається "поділитися" ключами з сусідом. Це призводить до того, що вузли довше залишаються більш заповненими (мінімум на 2/3), що краще використовує дисковий простір.

3. *a)* Мінімально дозволена кількість ключів у вузлі B-дерева порядку `t` — це `t-1`. Оскільки після видалення у вузлі залишилось `t-1=4` ключі, він все ще задовольняє інваріанту, і жодних додаткових дій з балансування не потрібно.

4. *c)* "Проактивне" злиття на шляху вниз означало б, що ми зливаємо вузли з `t-1` ключами, навіть якщо видалення відбудеться в іншому піддереві і це злиття було зайвим. Тому ефективніше спершу видалити, а потім локально виправляти проблему "недоповнення" на шляху назад.

5. *b)* Це ключова перевага B+-дерев. Зв'язаний список листків дозволяє дуже ефективно виконувати діапазонні запити. Знайшовши початкову точку за $O(\log_t N)$, подальший обхід є швидким лінійним проходом по дискових сторінках.

6. *c)* Кожен вузол має `2t-1` ключів та `2t` вказівників. Розмір вузла: $(2t-1)\cdot 8 + 2t \cdot 8 \approx 16t + 16t = 32t$. Ми хочемо, щоб $32t \leq 4096$, звідси $t \leq 128$. Обирають максимальне можливе значення.

7. *c)* Для точкових запитів (пошук за одним ключем) в оперативній пам'яті хеш-таблиця є асимптотично швидшою. Переваги B-дерева проявляються при роботі з диском та при виконанні діапазонних запитів.

8. *a)* Існує пряма відповідність (ізоморфізм) між цими двома структурами. 3-вузол можна представити як чорний батько з одним червоним нащадком, а 4-вузол — як чорний батько з двома червоними нащадками. Це пояснює, чому правила RBT працюють.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте клас B-дерева з порядком t=3 та операціями вставки і пошуку.

2. Ручне трасування. Вручну намалюйте, як буде змінюватися B-дерево порядку t=2 (тобто 2-3-4 дерево) при послідовній вставці ключів: 10, 20, 5, 15, 25, 35, 40.
3. Реалізація видалення. *(Завдання підвищеної складності)*. Додайте до вашого класу метод `delete`.
4. Аналіз висоти. Напишіть функцію, що обчислює висоту B-дерева. Порівняйте її з висотою збалансованого бінарного дерева для однакового набору з 1,000,000 елементів.
5. B+-дерево. Модифікуйте вашу реалізацію B-дерева, щоб вона відповідала правилам B+-дерева (дані лише в листках, листки зв'язані).
6. Діапазонний запит. Реалізуйте для вашого B+-дерева функцію, що ефективно знаходить усі елементи в заданому діапазоні `[min, max]`.
7. Есе-аналіз. Напишіть есе на тему "Чому B+-дерева, а не B-дерева, стали стандартом для індексів у реляційних базах даних?".
8. Візуалізатор. Створіть простий текстовий візуалізатор, що виводить структуру вашого B-дерева по рівнях.
9. Оцінка розміру. Розрахуйте, скільки максимум ключів типу `int` (4 байти) може зберігати B-дерево порядку t=128 висотою 2, якщо розмір вказівника — 8 байт.
10. Порівняння з хеш-індексом. Дослідіть та порівняйте B-tree index та Hash index у СУБД (напр., PostgreSQL). У яких випадках кожен з них є кращим вибором?

Дерево порядкових статистик

Розглядаючи, як можна розширити функціонал існуючих структур даних для вирішення нових, цікавих завдань. Ми вже знаємо, як швидко знаходити елементи за їхнім ключем. Але що, як нам потрібно швидко знайти елемент за його порядковим номером (рангом)? Для цього існують дерева порядкових статистик (Order-Statistic Trees, OST).

1. Теоретичні відомості

1.1. Мотивація: Пошук за рангом

Уявіть, що у вас є динамічна множина даних, наприклад, результати гравців у онлайн-грі. Звичайне збалансоване дерево пошуку (як AVL або червоно-чорне) дозволяє нам швидко виконувати такі операції:

- Вставити нового гравця з його балами.
- Видалити гравця.
- Знайти гравця за його іменем (ключем).
- Знайти гравця з мінімальними/максимальними балами.

Але що, як нам потрібно відповісти на питання: “Який гравець знаходиться на 50-му місці в рейтингу?” або “Яке місце в рейтингу займає гравець ‘Player123’?”

Звичайне дерево пошуку не може відповісти на ці питання ефективно. Щоб знайти 50-й елемент, нам доведеться виконати симетричний обхід і зупинитись на 50-му кроці, що в найгіршому випадку займе $O(N)$ часу. Щоб знайти ранг гравця, нам також доведеться обійти всі менші за нього елементи, що теж може зайняти $O(N)$ часу.

Дерево порядкових статистик — це не новий тип дерева, а розширена версія будь-якого збалансованого бінарного дерева пошуку (зазвичай червоно-чорного), яка дозволяє знаходити елемент за його рангом та ранг елемента за логарифмічний час.

1.2. Розширення: поле `size`

Магія полягає в додаванні до кожного вузла одного-єдиного додаткового поля: `size`.

`node.size` — це кількість вузлів у піддереві, коренем якого є `node` (включаючи сам `node`). Тобто, для будь-якого вузла `x` виконується рівність:

`x.size = x.left.size + x.right.size + 1`

Для зручності, ми вважаємо, що розмір `NULL` (листка-варти) дорівнює 0.

1.3. Підтримка поля `size`

Щоб ця інформація завжди була актуальною, нам потрібно оновлювати її при кожній операції, що змінює структуру дерева: вставці, видаленні та обертаннях.

Вставка:

OST - це модифікація червоно-чорного дерева, де кожен вузол зберігає додатковий атрибут **`size`** - кількість вузлів у піддереві з коренем у цьому вузлі.

Структура вузла OST

```
Node
{
    key: значення ключа
    color: RED або BLACK
    size: кількість вузлів у піддереві
    left: лівий нащадок
    right: правий нащадок
    parent: батьківський вузол
}
```

Операція вставки вузла

Крок 1: Стандартна вставка як у BST

Спочатку виконуємо звичайну вставку як у бінарному дереві пошуку:

12. Починаємо з кореня
13. Порівнюємо ключ нового вузла з поточним
14. Йдемо ліворуч, якщо ключ менший, праворуч - якщо більший
15. Повторюємо до знаходження вільного місця
16. Вставляємо новий вузол як лист

Важливо: Під час проходження по дереву збільшуємо **`size`** кожного вузла на шляху на 1.

Крок 2: Ініціалізація нового вузла

```
newNode.color = RED
newNode.size = 1
newNode.left = NIL
newNode.right = NIL
```

Крок 3: Оновлення розмірів

Після вставки треба оновити атрибут **`size`** для всіх предків нового вузла:

```
node = newNode.parent
while node != NIL:
    node.size = node.left.size + node.right.size + 1
    node = node.parent
```

Крок 4: Відновлення властивостей червоно-чорного дерева

Виконуємо стандартну процедуру `RB-INSERT-FIXUP` для відновлення властивостей червоно-чорного дерева. Критично важливо: під час поворотів треба правильно оновлювати значення `size`.

Оновлення size під час поворотів

Лівий поворот навколо вузла x:

```
y = x.right
x.right = y.left
y.left = x

// Оновлення розмірів
x.size = x.left.size + x.right.size + 1
y.size = y.left.size + y.right.size + 1
```

Правий поворот навколо вузла y:

```
x = y.left
y.left = x.right
x.right = y

// Оновлення розмірів
y.size = y.left.size + y.right.size + 1
x.size = x.left.size + x.right.size + 1
```

Псевдокод операції вставки

```
OST-INSERT(T, z):
    y = NIL
    x = T.root

    // Пошук місця для вставки та оновлення size
    while x != NIL:
        y = x
        x.size = x.size + 1  // Збільшуємо size на шляху
        if z.key < x.key:
            x = x.left
        else:
            x = x.right

    z.parent = y
    z.size = 1
    z.color = RED

    // Вставка вузла
    if y == NIL:
        T.root = z
    else if z.key < y.key:
```

```
        y.left = z
    else:
        y.right = z

    // Відновлення властивостей RB-дерева
    OST-INSERT-FIXUP(T, z)
```

Складність операції

- Часова складність: O(log n) - така ж, як у червоно-чорному дереві
- Просторова складність: O(1) - додаткова пам’ять не потрібна

Ключові моменти

1. Оновлення **`size`** відбувається під час спуску по дереву
2. Повороти вимагають перерахунку **`size`** для залучених вузлів
3. Властивості червоно-чорного дерева зберігаються
4. Після вставки дерево підтримує можливість швидкого пошуку за рангом та визначення рангу елемента

Така реалізація дозволяє виконувати операції `SELECT` (знайти i-й найменший елемент) та `RANK` (знайти ранг елемента) за час O(log n).

Видалення вузла

Операція видалення вузла в Order-Statistic Tree є найскладнішою, оскільки потрібно не лише видалити вузол та відновити властивості червоно-чорного дерева, але й коректно оновити атрибути **`size`** у всіх залучених вузлах.

Операція видалення складається з кількох етапів:

1. Зменшення **`size`** на шляху до вузла

2. Стандартне видалення з BST

3. Оновлення **`size`** після структурних змін

4. Відновлення властивостей RB-дерева з коректним оновленням **`size`**

Крок 1: Пошук вузла та оновлення size

При пошуку вузла для видалення зменшуємо **`size`** на шляху:

```
OST-SEARCH-AND-UPDATE(T, k):
    x = T.root
    while x != NIL and k != x.key:
        x.size = x.size - 1  // Зменшуємо size на шляху
```

```
        if k < x.key:
            x = x.left
        else:
            x = x.right

    if x != NIL:
        x.size = x.size - 1  // Зменшуємо size знайденого вузла

    return x
```

Крок 2: Класифікація випадків видалення

Випадок 1: Вузол не має дітей (лист)

```
if z.left == NIL and z.right == NIL:
    // Просто видаляємо вузол
    // Size вже оновлений на кроці 1
```

Випадок 2: Вузол має одну дитину

```
if z.left == NIL:
    child = z.right
else:
    child = z.left

// Замінюємо z на child
// Size вже оновлений на кроці 1
```

Випадок 3: Вузол має дві дитини

```
// Знаходимо наступника (successor)
y = TREE-SUCCESSOR(z)

// Копіюємо ключ наступника
z.key = y.key

// Видаляємо наступника (у нього максимум одна дитина)
// Потрібно оновити size на шляху до наступника
```

Крок 3: Детальний псевдокод операції видалення

```
OST-DELETE(T, z):
    original_color = z.color

    if z.left == NIL:
        x = z.right
        OST-TRANSPLANT(T, z, z.right)

    else if z.right == NIL:
        x = z.left
        OST-TRANSPLANT(T, z, z.left)
```

```
    else:
        y = TREE-MINIMUM(z.right)  // Знаходимо наступника
        original_color = y.color
        x = y.right

        if y.parent == z:
            x.parent = y
        else:
            OST-TRANSPLANT(T, y, y.right)
            y.right = z.right
            y.right.parent = y

            // Оновлюємо size від y до z.right
            UPDATE-SIZE-PATH(y.parent, z.right)

        OST-TRANSPLANT(T, z, y)
        y.left = z.left
        y.left.parent = y
        y.color = z.color

        // Оновлюємо size для y
        y.size = y.left.size + y.right.size + 1

    // Оновлюємо size для всіх предків
    UPDATE-SIZE-ANCESTORS(x.parent)

    if original_color == BLACK:
        OST-DELETE-FIXUP(T, x)
```

Крок 4: Допоміжні функції

Функція трансплантації з оновленням size

```
OST-TRANSPLANT(T, u, v):
    if u.parent == NIL:
        T.root = v
    else if u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v

    if v != NIL:
        v.parent = u.parent
```

Оновлення size для предків

```
UPDATE-SIZE-ANCESTORS(node):
```

```
    while node != NIL:
        node.size = node.left.size + node.right.size + 1
        node = node.parent
```

Оновлення size на шляху

```
UPDATE-SIZE-PATH(start, end):
    node = start
    while node != end:
        node.size = node.left.size + node.right.size + 1
        node = node.parent
```

Крок 5: Відновлення властивостей RB-дерева

Модифікована функція `OST-DELETE-FIXUP` з оновленням `size`:

```
OST-DELETE-FIXUP(T, x):
    while x != T.root and x.color == BLACK:
        if x == x.parent.left:
            w = x.parent.right

            if w.color == RED:
                w.color = BLACK
                x.parent.color = RED
                LEFT-ROTATE(T, x.parent)
                w = x.parent.right

            if w.left.color == BLACK and w.right.color == BLACK:
                w.color = RED
                x = x.parent
            else:
                if w.right.color == BLACK:
                    w.left.color = BLACK
                    w.color = RED
                    RIGHT-ROTATE(T, w)
                    w = x.parent.right

                w.color = x.parent.color
                x.parent.color = BLACK
                w.right.color = BLACK
                LEFT-ROTATE(T, x.parent)
                x = T.root
        else:
            // Симетричний випадок

    x.color = BLACK
```

Крок 6: Модифіковані обертання з оновленням size

Обертання. Детальний розгляд.

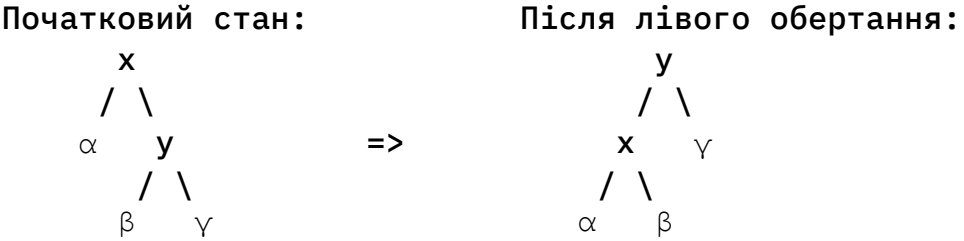
Операції обертання в Order-Statistic Tree є критично важливими, оскільки вони не лише змінюють структуру дерева, але й вимагають коректного оновлення атрибутів **size** для підтримки властивостей OST. Ці операції майже повністю співпадають з операціями обертання для червоно-чорного дерева. з єдиним доповненням - оновленням атрибуту size. Це демонструє елегантність підходу розширених структур даних, де базова логіка залишається незмінною, а додається лише підтримка додаткової інформації. Нагадаємо ці операції:

Типи обертань:

1. Ліве обертання (Left Rotation)

2. Праве обертання (Right Rotation)

Візуалізація лівого обертання



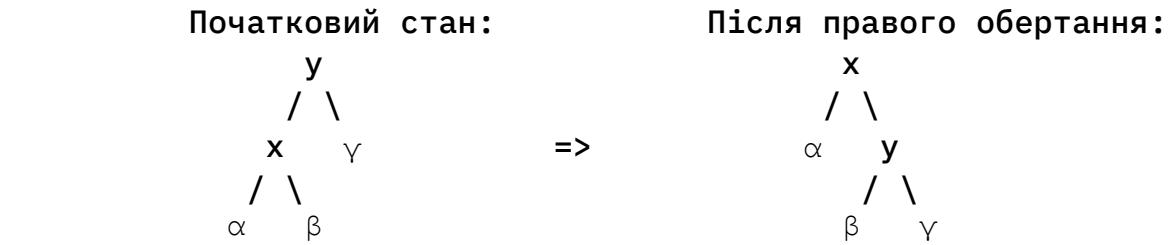
Псевдокод лівого обертання з оновленням **size:**

```
LEFT-ROTATE(T, x):
    // Перевіряємо, чи можливе обертання
    if x.right == NIL:
        return ERROR  // Неможливо виконати ліве обертання

    y = x.right        // y стане новим коренем піддерева

    // Крок 1: Переміщення β піддерева
    x.right = y.left
    if y.left != NIL:
        y.left.parent = x

    // Крок 2: Зв'язування y з батьком x
    y.parent = x.parent
    if x.parent == NIL:
        T.root = y     // y стає новим коренем дерева
    else if x == x.parent.left:
        x.parent.left = y
    else:
        x.parent.right = y

    // Крок 3: Встановлення x як лівої дитини y
```

```
    y.left = x
    x.parent = y

    // Крок 4: КРИТИЧНО - Оновлення size атрибутів
    x.size = x.left.size + x.right.size + 1
    y.size = y.left.size + y.right.size + 1
```

Праве обертання навколо вузла y

Візуалізація правого обертання



Псевдокод правого обертання з оновленням **size:**

```
RIGHT-ROTATE(T, y):
    // Перевіряємо, чи можливе обертання
    if y.left == NIL:
        return ERROR  // Неможливо виконати праве обертання

    x = y.left         // x стане новим коренем піддерева

    // Крок 1: Переміщення β піддерева
    y.left = x.right
    if x.right != NIL:
        x.right.parent = y

    // Крок 2: Зв'язування x з батьком y
    x.parent = y.parent
    if y.parent == NIL:
        T.root = x     // x стає новим коренем дерева
    else if y == y.parent.right:
        y.parent.right = x
    else:
        y.parent.left = x

    // Крок 3: Встановлення y як правої дитини x
    x.right = y
    y.parent = x

    // Крок 4: КРИТИЧНО - Оновлення size атрибутів
    y.size = y.left.size + y.right.size + 1
    x.size = x.left.size + x.right.size + 1
```

Проведемо математичне обґрунтування оновлення size.

Для лівого обертання:

До обертання: -

$x.size = size(\alpha) + size(\beta) + size(\gamma) + 1$ - $y.size = size(\beta) + size(\gamma) + 1$

Після обертання: -

$x.size = size(\alpha) + size(\beta) + 1$ - $y.size = size(\alpha) + size(\beta) + size(\gamma) + 1$

Для правого обертання:

До обертання: -

$y.size = size(\alpha) + size(\beta) + size(\gamma) + 1$ - $x.size = size(\alpha) + size(\beta) + 1$

Після обертання: -

$y.size = size(\beta) + size(\gamma) + 1$ - $x.size = size(\alpha) + size(\beta) + size(\gamma) + 1$

Ця рівність називається інваріантом операції обертання. Інваріанти в операціях обертання для порядкових дерев потрібні для підтримання коректності додаткової інформації, яка зберігається у вузлах дерева.

Основні інваріанти для операцій обертання:

1. Інваріант розміру: $size[x] = size[left[x]] + size[right[x]] + 1$
    - Розмір вузла дорівнює сумі розмірів лівого та правого піддерев плюс 1
2. Коректність BST: після обертання дерево має залишатися валідним бінарним деревом пошуку

Виконання інваріантів критично важливо, тому що при стандартному обертанні змінюється структура дерева, і поля **size** у деяких вузлах стають некоректними. Якщо не оновити ці значення, подальші операції пошуку k-го елемента або визначення рангу видаватимуть неправильні результати.

Без підтримання інваріантів дерево технічно залишиться BST, але втратить здатність ефективно виконувати операції, пов’язані з порядковими статистиками, що є основною метою використання таких структур даних.

Оптимізована версія операції обертання з інваріантами:

```
SAFE-LEFT-ROTATE(T, x):
    // Перевірка інваріантів
    assert(x != NIL)
    assert(x.right != NIL)

    y = x.right

    // Зберігаємо початкові розміри для перевірки
    old_x_size = x.size
    old_y_size = y.size

    // Структурні зміни
    x.right = y.left
    if y.left != NIL:
        y.left.parent = x

    y.parent = x.parent
    if x.parent == NIL:
        T.root = y
    else if x == x.parent.left:
        x.parent.left = y
    else:
        x.parent.right = y

    y.left = x
    x.parent = y

    // Оновлення size
    x.size = x.left.size + x.right.size + 1
    y.size = y.left.size + y.right.size + 1

    // Перевірка інваріанта: загальна кількість вузлів не змінилася
    assert(y.size == old_x_size)
```

Використання обертань у операціях OST

У операції вставки (INSERT-FIXUP):

```
OST-INSERT-FIXUP(T, z):
    while z.parent.color == RED:
        if z.parent == z.parent.parent.left:
            y = z.parent.parent.right
            if y.color == RED:
                // Випадок 1: Перефарбування
                z.parent.color = BLACK
```

```
                y.color = BLACK
                z.parent.parent.color = RED
                z = z.parent.parent
            else:
                if z == z.parent.right:
                    // Випадок 2: Ліве обертання
                    z = z.parent
                    LEFT-ROTATE(T, z)  // Автоматично оновлює size

                // Випадок 3: Праве обертання
                z.parent.color = BLACK
                z.parent.parent.color = RED
                RIGHT-ROTATE(T, z.parent.parent)  // Автоматично онов
лює size
        else:
            // Симетричний випадок

    T.root.color = BLACK
```

У операції видалення (DELETE-FIXUP):

```
OST-DELETE-FIXUP(T, x):
    while x != T.root and x.color == BLACK:
        if x == x.parent.left:
            w = x.parent.right

            if w.color == RED:
                w.color = BLACK
                x.parent.color = RED
                LEFT-ROTATE(T, x.parent)  // Оновлює size
                w = x.parent.right

            if w.left.color == BLACK and w.right.color == BLACK:
                w.color = RED
                x = x.parent
            else:
                if w.right.color == BLACK:
                    w.left.color = BLACK
                    w.color = RED
                    RIGHT-ROTATE(T, w)  // Оновлює size
                    w = x.parent.right

                w.color = x.parent.color
                x.parent.color = BLACK
                w.right.color = BLACK
                LEFT-ROTATE(T, x.parent)  // Оновлює size
```

```
                x = T.root
        else:
            // Симетричний випадок

    x.color = BLACK
```

Складність операцій обертання

- Часова складність: O(1) для одного обертання
- Просторова складність: O(1)

Критичні моменти при реалізації:

1. Порядок оновлення size

```
// ПРАВИЛЬНО: Спочатку оновлюємо нижчий вузол
x.size = x.left.size + x.right.size + 1
y.size = y.left.size + y.right.size + 1

// НЕПРАВИЛЬНО: Неправильний порядок може призвести до некоректних зн
ачень
y.size = y.left.size + y.right.size + 1
x.size = x.left.size + x.right.size + 1
```

2. Обробка NULL вузлів

```
// Завжди перевіряємо на NULL
if y.left != NULL:
    y.left.parent = x

// Розмір NIL вузла завжди 0
NIL.size = 0
```

3. Інваріанти OST

Після кожного обертання мають зберігатися: - Властивості BST - Властивості червоно-чорного дерева - Коректність атрибутів **`size`**

Тестування коректності

```
VERIFY-OST-PROPERTIES(node):
    if node == NIL:
        return true

    // Перевіряємо коректність size
    expected_size = node.left.size + node.right.size + 1
    if node.size != expected_size:
        return false

    // Рекурсивно перевіряємо піддерева
```

```
    return VERIFY-OST-PROPERTIES(node.left) and
           VERIFY-OST-PROPERTIES(node.right)
```

Обертання є фундаментальною операцією в OST, яка забезпечує ефективне балансування дерева при збереженні всіх необхідних властивостей та коректних значень `size` атрибутів.

1.4. Нові операції: `Select` та `Rank`

OS-Select: Пошук елемента за рангом

Функція `OS-Select(node, i)` знаходить вузол з `i`-м рангом (тобто, `i`-й найменший елемент у дереві). Ранги нумеруються з 1.

Логіка:

1. Починаємо з кореня. Для поточного вузла `x` обчислюємо ранг його кореня всередині його власного піддерева. Цей ранг дорівнює `r = x.left.size + 1`.
2. Порівнюємо шуканий ранг `i` з `r`:
    - Якщо `i == r`, то поточний вузол `x` і є тим, що ми шукаємо.
    - Якщо `i < r`, то шуканий елемент знаходиться у лівому піддереві. Рекурсивно шукаємо в `x.left` елемент з тим самим рангом `i`.
    - Якщо `i > r`, то шуканий елемент знаходиться у правому піддереві. Але ми вже “пропустили” `r` елементів (все ліве піддерево і корінь). Тому в правому піддереві ми будемо шукати елемент з рангом `i - r`.

Псевдокод такий:

```
ФУНКЦІЯ OS_Select(node, i):
    // Перевірка на валідність рангу
    ЯКЩО i > node.size АБО i <= 0 ТО ПОВЕРНУТИ NULL КІНЕЦЬ ЯКЩО

    // Обчислюємо ранг кореня поточного піддерева
    left_size = (node.left != NULL) ? node.left.size : 0
    r = left_size + 1

    // Порівнюємо і приймаємо рішення
    ЯКЩО i == r ТО
        ПОВЕРНУТИ node
```

```
    ІНАКШЕ ЯКЩО i < r ТО
        ПОВЕРНУТИ OS_Select(node.left, i)
    ІНАКШЕ // i > r
        ПОВЕРНУТИ OS_Select(node.right, i - r)
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ
```

OS-Rank: Пошук рангу для елемента

Функція `OS-Rank(tree, x)` знаходить ранг вузла `x` (його позицію у відсортованій послідовності).

Логіка:

3. Починаємо з рангу, що дорівнює розміру лівого піддерева вузла `x` плюс один: `rank = x.left.size + 1`.
4. Починаємо підніматись вгору від `x` до кореня. На кожному кроці:
    - Якщо ми щойно прийшли з правого піддерева (тобто `x` був правим нащадком свого батька `p`), це означає, що ми маємо додати до нашого рангу всі елементи з лівого піддерева `p` та сам вузол `p`. Тобто `rank = rank + p.left.size + 1`.
    - Якщо ми прийшли з лівого піддерева, ранг не змінюється.
5. Продовжуємо, доки не дійдемо до кореня.

Псевдокод такий:

```
ФУНКЦІЯ OS_Rank(tree, x):
    // Початковий ранг відносно власного піддерева
    rank = (x.left != NULL) ? x.left.size : 0
    rank = rank + 1

    // Піднімаємось до кореня
    y = x
    ПОКИ y != tree.root:
        // Якщо y є правим нащадком
        ЯКЩО y == y.parent.right ТО
            parent_left_size = (y.parent.left != NULL) ? y.parent.lef
t.size : 0
            rank = rank + parent_left_size + 1
        КІНЕЦЬ ЯКЩО
        y = y.parent
    КІНЕЦЬ ПОКИ
```

```
    ПОВЕРНУТИ rank
КІНЕЦЬ ФУНКЦІЇ
```

2. Область застосування

- Системи рейтингів та лідерборди: Швидке визначення місця гравця та гравця на N-му місці.
- Аналіз даних: Швидкий пошук медіани (N/2-ий елемент), перцентилів та інших порядкових статистик у динамічних наборах даних.
- Обчислювальна геометрія: Для задач, що вимагають швидкого доступу до елементів за їхнім порядком.

3. Приклади застосування

1. **Задача:** Система рейтингів у грах

Онлайн гра з мільйонами гравців потребує швидкого визначення рейтингу гравця та пошуку гравців за рейтингом.

Операції:

```
# Додавання нового гравця
ost.insert(player_id, rating)

# Знаходження рейтингу гравця (позиція в загальному списку)
rank = ost.rank(player_id)
print(f"Гравець {player_id} займає {rank} місце")

# Знаходження топ-10 гравців
top_players = []
for i in range(1, 11):
    player = ost.select(i)  # i-й найкращий гравець
    top_players.append(player)

# Знаходження гравців у певному діапазоні рейтингів
players_in_range = ost.range_query(1200, 1500)
Переваги OST:
```

- Вставка нового гравця: O(log n)
- Визначення рейтингу: O(log n)
- Пошук топ-k: O(k log n)
- Альтернатива (масив): O(n) для вставки

2. **Задача:** Медіана у потоці даних

Система моніторингу отримує потік числових даних та потребує постійного обчислення медіани.

Реалізація:

```
class MedianFinder:
    def __init__(self):
        self.ost = OrderStatisticTree()
        self.size = 0

    def add_number(self, num):
        self.ost.insert(num)
        self.size += 1

    def find_median(self):
        if self.size % 2 == 1:
            # Непарна кількість елементів
            return self.ost.select(self.size // 2 + 1)
        else:
            # Парна кількість елементів
            left = self.ost.select(self.size // 2)
            right = self.ost.select(self.size // 2 + 1)
            return (left + right) / 2
```

Приклад використання:

```
mf = MedianFinder()
data_stream = [5, 15, 1, 3, 7, 9, 12]

for num in data_stream:
    mf.add_number(num)
    print(f"Додано {num}, медіана: {mf.find_median()}")
```

9. Задача: Система управління таксопарком

Таксопарк потребує швидкого призначення найближчого доступного таксі клієнту.

Реалізація:

```
class TaxiDispatcher:
    def __init__(self):
        # OST для доступних таксі (сортування за відстанню)
        self.available_taxis = OrderStatisticTree()
        # OST для зайнятих таксі (сортування за часом завершення)
        self.busy_taxis = OrderStatisticTree()
```

```
    def add_taxi(self, taxi_id, current_location):
        distance = calculate_distance(current_location, client_locati
on)
        self.available_taxis.insert(taxi_id, distance)

    def assign_nearest_taxi(self, client_location):
        # Знаходимо найближче таксі (1-й елемент)
        nearest_taxi = self.available_taxis.select(1)

        # Переміщуємо до зайнятих
        self.available_taxis.delete(nearest_taxi.id)
        estimated_finish = current_time + trip_duration
        self.busy_taxis.insert(nearest_taxi.id, estimated_finish)

        return nearest_taxi

    def get_top_k_nearest(self, k):
        # Отримуємо k найближчих таксі
        return [self.available_taxis.select(i) for i in range(1, k+1)
]
```

4. Задача: Система торгівлі цінними паперами.

   Біржа потребує швидкого виконання ордерів з пріоритетом за ціною та часом.

Реалізація:

```
class StockExchange:
    def __init__(self):
        # OST для ордерів на купівлю (сортування за ціною убуванням)
        self.buy_orders = OrderStatisticTree()
        # OST для ордерів на продаж (сортування за ціною зростанням)
        self.sell_orders = OrderStatisticTree()

    def place_buy_order(self, order_id, price, volume):
        # Використовуємо негативну ціну для сортування убуванням
        self.buy_orders.insert(order_id, -price, volume)
        self.try_match_orders()

    def place_sell_order(self, order_id, price, volume):
        self.sell_orders.insert(order_id, price, volume)
        self.try_match_orders()

    def try_match_orders(self):
        # Найвища ціна купівлі (1-й елемент)
        best_buy = self.buy_orders.select(1)
```

```
        # Найнижча ціна продажу (1-й елемент)
        best_sell = self.sell_orders.select(1)

        if best_buy and best_sell and abs(best_buy.price) >= best_sel
l.price:
            # Виконуємо угоду
            self.execute_trade(best_buy, best_sell)

    def get_order_book_depth(self, levels=5):
        # Отримуємо топ-5 ордерів на купівлю та продаж
        buy_levels = [self.buy_orders.select(i) for i in range(1, lev
els+1)]
        sell_levels = [self.sell_orders.select(i) for i in range(1, l
evels+1)]
        return buy_levels, sell_levels
```

5. Задача: Система рекомендацій у соціальній мережі

Соціальна мережа потребує швидкого знаходження найбільш активних користувачів для рекомендацій.

Реалізація:

```
class SocialNetworkRecommender:
    def __init__(self):
        # OST для користувачів (сортування за рівнем активності)
        self.users_by_activity = OrderStatisticTree()
        # OST для постів (сортування за популярністю)
        self.posts_by_popularity = OrderStatisticTree()

    def update_user_activity(self, user_id, activity_score):
        # Оновлюємо рівень активності користувача
        self.users_by_activity.delete(user_id)
        self.users_by_activity.insert(user_id, activity_score)

    def get_top_active_users(self, count=10):
        # Отримуємо найактивніших користувачів
        return [self.users_by_activity.select(i) for i in range(1, co
unt+1)]

    def get_user_activity_rank(self, user_id):
        # Визначаємо рейтинг активності користувача
        return self.users_by_activity.rank(user_id)

    def recommend_users_in_range(self, user_id, similarity_threshold=
100):
        # Знаходимо користувачів з подібним рівнем активності
```

```
        user_rank = self.users_by_activity.rank(user_id)

        recommendations = []
        # Користувачі вище за рейтингом
        for i in range(max(1, user_rank - similarity_threshold), user
_rank):
            recommendations.append(self.users_by_activity.select(i))

        # Користувачі нижче за рейтингом
        for i in range(user_rank + 1, min(self.users_by_activity.size
, user_rank + similarity_threshold)):
            recommendations.append(self.users_by_activity.select(i))

        return recommendations
```

Переваги та недоліки

Переваги:

- Надзвичайно швидкі операції **`Select`** та **`Rank`**: Обидві виконуються за час $O(h)$, що для збалансованого дерева дорівнює $O(\log N)$.
- Незначне ускладнення: Додавання та підтримка поля **`size`** лише на константний фактор уповільнює стандартні операції вставки та видалення, не змінюючи їхньої асимптотичної складності.

Недоліки:

- Додаткова пам'ять: Кожен вузол вимагає додаткової пам'яті для зберігання поля **`size`**.
- Ускладнення реалізації: Потрібно коректно обробляти оновлення поля **`size`** при всіх модифікуючих операціях, особливо при обертаннях.

Контрольні запитання

1. Яку додаткову інформацію зберігає вузол дерева порядкових статистик ?
2. Як обчислюється значення поля **`size`** для вузла **`x`**?
3. Поясніть логіку роботи операції **`OS-Select(i)`**. Чому при переході у праве піддерево ми змінюємо шуканий ранг **`i`**?
4. На якій базовій структурі даних зазвичай будують дерево порядкових статистик і чому?

5. Як змінюються поля **`size`** вузлів **`x`** та **`y`** після виконання лівого обертання **`leftRotate(x)`**?

Методичні вказівки до лабораторної роботи

1. Назва

Розширення червоно-чорного дерева для підтримки порядкових статистик.

2. Мета

Практичне засвоєння техніки розширення структур даних. Реалізація операцій **`OS-Select`** та **`OS-Rank`**.

3. Завдання

Взяти існуючу реалізацію червоно-чорного дерева та модифікувати її, щоб вона стала деревом порядкових статистик .

4. Опис

Потрібно додати до структури вузла поле **`size`**. Модифікувати операції вставки, видалення та обертання для коректної підтримки цього поля. Реалізувати нові функції **`OS-Select`** та **`OS-Rank`**.

5. Порядок виконання

1. Додайте поле **`size`** до вашого класу **`Node`**. Ініціалізуйте його значенням 1 для нових вузлів.
2. Модифікуйте обертання: У функціях **`leftRotate`** та **`rightRotate`** додайте логіку для перерахунку полів **`size`** двох вузлів, що беруть участь в обертанні.
3. Модифікуйте вставку та видалення: Додайте логіку інкременту/декременту поля **`size`** для вузлів на шляху до кореня.
4. Реалізуйте **`OS-Select`**: Напишіть рекурсивну функцію для пошуку i-го елемента.
5. Реалізуйте **`OS-Rank`**: Напишіть функцію для пошуку рангу елемента.
6. Протестуйте: Перевірте коректність роботи всіх операцій.

6. Результат

Додаток, що реалізує дерево порядкових статистик з усіма основними операціями.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми.
2. Продемонструвати роботу операцій `OS-Select` та `OS-Rank` на прикладах.
3. Бути готовим відповісти на питання:
    - Як саме ви оновлюєте поле `size` при обертанні?
    - Пояснити рекурсивну логіку функції `OS-Select`.
    - Яка складність усіх реалізованих вами операцій?

Тести

1. Яке поле додається до вузла збалансованого дерева, щоб перетворити його на дерево порядкових статистик ?

a) `height`

b) `color`

c) `priority`

d) `size`

2. Чому дорівнює `node.size` для вузла `node`?

a) Кількість його нащадків.

b) Кількість вузлів у його лівому піддереві.

c) Кількість вузлів у піддереві з коренем `node`, включаючи сам `node`.

d) Висота піддерева.

3. Яка часова складність операції `OS-Select` у дереві порядкових статистик з N елементами?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(N \log N)$

4. На якій структурі даних найчастіше будують дерево порядкових статистик ?

a) На простому бінарному дереві пошуку.

b) На червоно-чорному дереві.

c) На масиві.

d) На хеш-таблиці.

5. При виконанні `OS-Select(node, i)` ми обчислюємо ранг кореня `r = node.left.size + 1`. Якщо `i > r`, що ми робимо далі?

a) Шукаємо ранг `i` у правому піддереві.

b) Шукаємо ранг `i+r` у правому піддереві.

c) Шукаємо ранг `i-r` у правому піддереві.

d) Повертаємо помилку.

Відповіді на прості тести: 1-d, 2-c, 3-b, 4-b, 5-c.

Тести підвищеної складності

1. Під час якої операції з червоно-чорним деревом оновлення поля `size` є найскладнішим?

a) Пошук

b) Знаходження мінімуму

c) Вставка

d) Обертання

2. Якщо ми хочемо знайти медіану в дереві порядкових статистик з N елементами, яку операцію ми викличемо?

a) `OS-Rank(N/2)`

b) `OS-Select(root, N/2)`

c) `Search(N/2)`

d) `OS-Rank(root, N/2)`

3. Яка просторова складність (додаткова пам'ять) дерева порядкових статистик порівняно з базовим червоно-чорним деревом?

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(N \log N)$

4. Якщо для вузла `x` `OS-Rank(tree, x)` повернула значення `k`, що поверне `OS-Select(tree.root, k)`?

a) Наступника `x`.

b) Попередника `x`.

c) Сам вузол `x`.

d) Корінь дерева.

5. Чи можна побудувати дерево порядкових статистик на основі AVL-дерева?

a) Ні, це неможливо.

b) Так, і це буде ефективніше, ніж на RBT.

c) Так, але логіка оновлення поля `size` при обертаннях буде такою ж, як і для RBT.

d) Так, але `OS-Select` буде працювати за $O(N)$.

6. Якщо ми викликали `OS-Select(node, i)` і `i > node.size`, що це означає?

a) Елемент знаходиться у правому піддереві.

b) Елемент знаходиться у лівому піддереві.

c) Запит є некоректним, оскільки ранг `i` виходить за межі кількості елементів у піддереві.

d) Це означає, що `node` є шуканим елементом.

7. Як знайти кількість елементів у діапазоні `[x, y]` за допомогою дерева порядкових статистик ?

a) `OS-Rank(y) - OS-Rank(x) + 1`

b) `OS-Select(y) - OS-Select(x)`

c) `OS-Rank(x) + OS-Rank(y)`

d) Це неможливо зробити ефективно.

8. Чому для реалізації `OS-Rank` простіше підніматися від вузла до кореня, а не спускатися від кореня?

a) Це швидше асимптотично.

b) Спуск неможливий, оскільки ми не знаємо, де знаходиться вузол `x`.

c) Спуск вимагає знання ключів, а підйом — лише структури (хто є чиїм батьком), що робить логіку простішою.

d) Обидва підходи однаково складні.

Відповіді.

1. *d)* Обертання повністю змінює локальну структуру дерева (батько стає нащадком), тому поля **`size`** для двох вузлів потрібно повністю перерахувати на основі нових розмірів їхніх нащадків, а не просто інкрементувати.

2. *b)* Медіана — це елемент, що знаходиться посередині відсортованої послідовності. Його ранг — `N/2`. Операція `OS-Select` знаходить елемент за його рангом.

3. *c)* Ми додаємо одне додаткове поле (**`size`**) до кожного з N вузлів дерева. Тому загальні додаткові витрати пам'яті є лінійними відносно кількості елементів.

4. *c)* Ці дві операції є оберненими одна до одної. `OS-Rank` знаходить ранг вузла, а `OS-Select` за цим рангом знаходить сам вузол.

5. *c)* Техніка розширення є загальною. Її можна застосувати до будь-якого збалансованого дерева, що використовує обертання. Логіка підтримки поля **`size`** при обертаннях не залежить від того, чому саме це обертання було викликане (правилами AVL чи RBT).

6. *c)* Ранг не може бути більшим за кількість елементів у множині (або підмножині, яку ми розглядаємо). Це свідчить про логічну помилку в запиті.

7. *a)* Знайшовши ранг більшого елемента `y` і меншого `x`, ми можемо обчислити кількість елементів між ними. Обидві операції `OS-Rank` виконуються за $O(\log N)$, тому і вся операція є дуже швидкою.

8. *c)* При підйомі від `x` логіка дуже проста: якщо ми прийшли з правого нащадка, ми додаємо розмір лівого піддерева + 1. Спроба реалізувати це через спуск вимагатиме більш складної логіки порівняння ключів на кожному кроці.

Завдання для самостійної роботи

1. Реалізація. Повністю реалізуйте розширення червоно-чорного дерева для підтримки порядкових статистик.

2. Ручне трасування. Вручну намалюйте дерево порядкових статистик після вставки чисел 10, 5, 15, 3, 7, 12, 18. Вкажіть поле `size` для кожного вузла. Протрасуйте `OS-Select(root, 4)` та `OS-Rank` для вузла з ключем 12.

3. Видалення за рангом. Напишіть функцію `OS-Delete(i)`, яка видаляє з дерева i-й за порядком елемент.

4. Пошук за ключем. Чи можна реалізувати `OS-Rank` через спуск від кореня, а не підйом від вузла? Напишіть псевдокод для такої версії.

5. Розширення для сум. Модифікуйте дерево так, щоб кожен вузол зберігав суму всіх ключів у своєму піддереві. Реалізуйте функцію, що знаходить суму ключів у діапазоні `[x, y]` за $O(\log N)$.

6. Есе-аналіз. Напишіть есе на тему “Техніка розширення структур даних. Наведіть 3-4 приклади, де вона може бути корисною, окрім порядкових статистик”.

7. Ітеративна реалізація. Напишіть ітеративну (нерекурсивну) версію функції `OS-Select`.

8. Декартове дерево. Чи можна розширити Декартове дерево для підтримки порядкових статистик? Які складнощі виникнуть?

9. Медіана потоку. Опишіть алгоритм, який, використовуючи два дерева порядкових статистик , може підтримувати медіану потоку чисел за час $O(\log N)$ на кожен новий елемент.

10. Багатовимірні дані. Дослідіть, як ідеї порядкових статистик застосовуються у багатовимірних структурах даних, як-от k-d дерева.

Лабораторні роботи: Order-Statistic Trees

Завдання охоплюють:

Базовий рівень (1-4): - Реалізація основних операцій OST - Операції SELECT та RANK - Практичні застосування (рейтинги, медіана)

Середній рівень (5-8): - Складніші системи (черга пріоритетів, біржа) - Порівняльний аналіз ефективності - Рекомендаційні системи

Просунутий рівень (9-12): - Статистичний аналіз великих даних - Системи реального часу - Географічні та моніторингові системи

Кожна лабораторна робота включає: - Чіткі технічні вимоги - Конкретні тестові дані - Очікувані результати - Критерії оцінювання

Лабораторна робота №1: Базова реалізація OST

Завдання:

Реалізувати базову структуру Order-Statistic Tree з операціями вставки, видалення, пошуку та обертання.

Технічні вимоги:

- Структура вузла з полями: key, color, size, left, right, parent
- Функції: INSERT, DELETE, SEARCH, LEFT-ROTATE, RIGHT-ROTATE
- Автоматичне оновлення атрибуту size при всіх операціях

Тестові дані:

```
Вставити послідовність: [15, 10, 20, 8, 12, 17, 25, 6, 11, 13, 22, 27
]
Видалити елементи: [10, 20, 17]
Перевірити коректність size для всіх вузлів
```

Очікуваний результат:

- Валідне OST дерево з правильними значеннями size
- Збереження властивостей червоно-чорного дерева

– Тестування на 100+ елементах

Лабораторна робота №2: Операції SELECT та RANK

Завдання:

Реалізувати операції SELECT (знаходження k-го найменшого елемента) та RANK (визначення позиції елемента).

Технічні вимоги:

– Функція SELECT(T, k) - повертає k-й найменший елемент

– Функція RANK(T, x) - повертає ранг елемента x

– Часова складність: O(log n) для обох операцій

Тестові завдання:

```
Дерево: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
1. SELECT(T, 5) = ?
2. RANK(T, 13) = ?
3. SELECT(T, 1) = ?
4. RANK(T, 19) = ?
```

Додаткове завдання:

Знайти медіану дерева використовуючи операцію SELECT.

Лабораторна робота №3: Система рейтингів гравців

Завдання:

Розробити систему рейтингів для онлайн-гри з можливістю швидкого визначення позиції гравця та топ-списків.

Функціональність:

– Додавання нового гравця з рейтингом

– Оновлення рейтингу існуючого гравця

– Знаходження позиції гравця в загальному рейтингу

– Отримання топ-N гравців

– Знаходження гравців у певному діапазоні рейтингів

Вхідні дані:

```
Гравці: [(Player1, 1500), (Player2, 2000), (Player3, 1200),
         (Player4, 1800), (Player5, 1600)]
Операції:
```

```
- Додати (Player6, 1750)
- Знайти ранг Player3
- Отримати топ-3 гравців
- Оновити рейтинг Player1 до 2100
```
Вимоги до виводу:

Детальний звіт про всі операції та стан рейтингової таблиці.

Лабораторна робота №4: Динамічне обчислення медіани

Завдання:

Реалізувати систему для обчислення медіани в потоці даних, що надходять у реальному часі.

Порядок виконання:

- Підтримувати OST для всіх отриманих значень
- При додаванні нового елемента оновлювати медіану
- Підтримувати історію медіан

Вхідні дані:

```
Потік даних: [5, 15, 1, 3, 7, 9, 12, 4, 8, 11, 6, 10]
Після кожного додавання вивести поточну медіану
```
Розширене завдання:

Реалізувати обчислення довільних персентилів (25-й, 75-й, 90-й).

Лабораторна робота №5: Система управління чергою пріоритетів

Завдання:

Створити систему управління чергою в лікарні з різними рівнями пріоритету пацієнтів.

Пріоритети:

- Критичний (1) - негайне обслуговування
- Терміновий (2) - обслуговування протягом години
- Звичайний (3) - обслуговування в порядку черги

Функціональність:

- Додавання пацієнта з пріоритетом та часом прибуття
- Визначення наступного пацієнта для обслуговування

– Пошук позиції пацієнта в черзі

– Статистика черги за пріоритетами

Тестовий сценарій:

```
Додати пацієнтів:
- (ID: 001, Пріоритет: 3, Час: 09:00)
- (ID: 002, Пріоритет: 1, Час: 09:15)
- (ID: 003, Пріоритет: 2, Час: 09:10)
- (ID: 004, Пріоритет: 3, Час: 09:05)
```

Лабораторна робота №6: Система торгівлі цінними паперами

Завдання:

Розробити спрощену систему біржової торгівлі з книгою ордерів (order book).

Компоненти:

– OST для ордерів на купівлю (сортування за ціною по спаданню)

– OST для ордерів на продаж (сортування за ціною по зростанню)

– Механізм зіставлення ордерів

Операції:

– Розміщення ордера на купівлю/продаж

– Скасування ордера

– Виконання угод при збігу цін

– Отримання найкращих цін (bid/ask)

– Глибина ринку (top-N цін)

Вхідні дані:

```
Ордери:
BUY: [(ID1, 100, 50), (ID2, 95, 30), (ID3, 105, 20)]
SELL: [(ID4, 110, 40), (ID5, 105, 25), (ID6, 115, 35)]
```

Лабораторна робота №7: Аналіз ефективності алгоритмів

Завдання:

Порівняти ефективність OST з іншими структурами даних для різних операцій.

Структури для порівняння:

- Order-Statistic Tree
- Відсортований масив
- Бінарна купа (heap)
- Звичайне BST

Операції для тестування:

- Вставка N елементів
- Пошук k-го найменшого елемента
- Видалення довільного елемента
- Визначення рангу елемента

Розміри даних:

N = 1000, 10000, 100000

Результат:

Графіки та таблиці порівняння часу виконання операцій.

Лабораторна робота №8: Система рекомендацій

Завдання:

Створити систему рекомендацій для соціальної мережі на основі рівня активності користувачів.

Метрики активності:

- Кількість постів
- Кількість коментарів
- Кількість вподобань
- Час онлайн

Функціональність:

- Обчислення загального рейтингу активності
- Пошук топ-K активних користувачів
- Рекомендації схожих за активністю користувачів
- Кластеризація користувачів за рівнем активності

Тестові дані:

```
Користувачі: 1000 записів з випадковими метриками активності
Операції:
- Знайти топ-10 активних користувачів
- Для користувача ID=500 знайти 5 схожих
- Розділити на 4 групи за активністю
```

Лабораторна робота №9: Статистичний аналіз даних

Завдання:

Розробити систему для обчислення різних статистичних показників великого набору даних.

Статистики для обчислення:

- Медіана, квартилі (Q1, Q3)
- Персентилі (10, 25, 50, 75, 90, 95, 99)
- Міжквартильний розмах (IQR)
- Виявлення викидів (outliers)

Функціональність:

- Завантаження великого масиву даних
- Динамічне оновлення статистик при додаванні нових даних
- Візуалізація розподілу даних
- Експорт статистичного звіту

Вхідні дані:

```
Набір з 10000 випадкових чисел (нормальний розподіл)
+ 500 додаткових значень, що додаються поступово
```

Лабораторна робота №10: Система управління ресурсами

Завдання:

Створити систему розподілу ресурсів (CPU, пам’ять) між процесами в операційній системі.

Критерії розподілу:

- Пріоритет процесу
- Час очікування

- Кількість необхідних ресурсів
- Тип процесу (системний/користувацький)

Алгоритм:

- Комбінований скор для кожного процесу
- Вибір наступного процесу для виконання
- Оновлення пріоритетів з часом
- Запобігання голодуванню (starvation)

Сценарій тестування:

```
50 процесів з різними характеристиками
Симуляція 1000 тактів планувальника
Збір статистики ефективності
```

Лабораторна робота №11: Географічна система пошуку

Завдання:

Розробити систему для пошуку найближчих об’єктів на основі відстані.

Компоненти:

- OST для зберігання об’єктів за відстанню від поточної позиції
- Динамічне оновлення при зміні позиції
- Фільтрація за типом об’єкта

Функціональність:

- Додавання нових об’єктів (ресторани, магазини, банки)
- Пошук K найближчих об’єктів певного типу
- Обновлення позиції користувача
- Пошук об’єктів в радіусі R

Вхідні дані:

```
Поточна позиція: (50.4501, 30.5234) - Київ
Об'єкти: 1000 точок з координатами та типами
Операції:
- Знайти 5 найближчих ресторанів
- Знайти всі банки в радіусі 1 км
- Змінити позицію та повторити пошук
```

Лабораторна робота №12: Система моніторингу продуктивності

Завдання:

Створити систему реального часу для моніторингу продуктивності веб-сервера.

Метрики:

- Час відповіді (response time)
- Кількість запитів за секунду (RPS)
- Використання пам’яті
- Завантаженість CPU

Функціональність:

- Збір метрик кожну секунду
- Обчислення SLA показників (95-й, 99-й персентиль)
- Виявлення аномалій
- Алерти при перевищенні порогів
- Історичний аналіз

Алгоритм виявлення аномалій:

```
Якщо поточне значення > 95-го персентиля за останні 10 хвилин:
    Сигнал попередження
Якщо поточне значення > 99-го персентиля за останні 10 хвилин:
    Критичний алерт
```

Симуляція:

```
Генерація 3600 вимірювань (1 година)
Включення кількох пікових навантажень
Демонстрація роботи системи алертів
```

Загальні вимоги до всіх лабораторних робіт:

Технічні вимоги:

- Код має бути добре документованим
- Обробка помилок та граничних випадків
- Тестування на різних розмірах даних
- Аналіз часової складності

Звітність:

- Опис алгоритму та структури даних
- Блок-схема або псевдокод
- Результати тестування
- Аналіз ефективності
- Висновки

Рівні складності:

- Базовий: Реалізація основного функціоналу
- Середній: Оптимізація та додаткові функції
- Просунутий: Інтеграція з іншими алгоритмами, візуалізація

Критерії оцінювання:

- Коректність реалізації (40%)
- Ефективність алгоритму (25%)
- Якість коду та документація (20%)
- Тестування та аналіз (15%)

Купи та пріоритетні черги

Досі ми розглядали структури даних, основною метою яких був ефективний пошук елемента за його ключем. Бінарні дерева пошуку, AVL, червоно-чорні, B-дерева — всі вони оптимізовані для того, щоб швидко відповідати на питання “чи є елемент X у множині?” або “які елементи знаходяться в діапазоні [Y, Z]?”.

Але існує цілий клас завдань, де нас цікавить зовсім інше питання: “Який елемент є найважливішим (або пріоритетним) на даний момент?”

Уявіть собі такі сценарії:

- Приймальне відділення лікарні: Який пацієнт потребує уваги лікаря в першу чергу?

- Операційна система: Яке з сотень завдань, що очікують на виконання, має отримати доступ до процесора наступним?

-Мережевий маршрутизатор: Який з пакетів даних у черзі має бути відправлений першим?

У цих випадках нам не потрібно шукати конкретного пацієнта чи завдання. Нам потрібно швидко знаходити та вилучати елемент з найвищим пріоритетом.

Для вирішення таких завдань існує абстрактний тип даних (АТД), який називається пріоритетна черга (Priority Queue). На відміну від звичайної черги, де працює принцип “першим прийшов — першим вийшов” (FIFO), або стека (“останнім прийшов — першим вийшов”, LIFO), пріоритетна черга працює за принципом “найбільш пріоритетний — вийшов першим”.

Вона підтримує, як мінімум, дві основні операції:

1. **`insert(element, priority)`**: Додати новий елемент з певним пріоритетом.

2. **`extract_max()`** (або **`extract_min()`**): Знайти, повернути та видалити елемент з найвищим (або найнижчим) пріоритетом.

Перейдемо до розгляду структури данних, яка реалізує цю філософію.

Купа: Ефективна реалізація

Тепер виникає питання: як реалізувати пріоритетну чергу ефективно? Можна було б використовувати звичайний масив, але тоді **`extract_max`** вимагатиме повного перебору ($O(N)$), або відсортований масив, але тоді **`insert`** буде повільним ($O(N)$). Збалансовані дерева пошуку можуть

виконати обидві операції за $O(\log N)$, але вони є надто складними для такого простого завдання.

Найбільш поширеною та ефективною структурою даних для реалізації пріоритетної черги є купа (Heap).

Купа — це спеціалізована деревоподібна структура даних, яка задовольняє одній простій, але потужній властивості купи (heap property): ключ у будь-якому вузлі є більшим або рівним (для max-heap) за ключі всіх його нащадків. Ця властивість гарантує, що елемент з найвищим пріоритетом завжди знаходиться в корені дерева, і доступ до нього є миттєвим ($O(1)$).

У цьому розділі ми детально розглянемо:

- Властивості бінарних куп та їхню геніальну реалізацію на основі звичайного масиву.
- Алгоритми для підтримки властивості купи.
- Реалізацію всіх операцій пріорітетної черги з логарифмічною складністю.

Бінарна купа

Бінарна купа є однією з найважливіших структур даних у комп'ютерних науках, що поєднує ефективність операцій з простотою реалізації. Ця структура даних знаходить широке застосування в алгоритмах сортування, пошуку найкоротших шляхів у графах, планування задач в операційних системах та багатьох інших областях. Бінарна купа забезпечує логарифмічний час виконання основних операцій при збереженні лінійної складності побудови з довільного масиву.

У цьому розділі ми детально розглянемо теоретичні основи бінарної купи, математичні принципи, що лежать в її основі, алгоритми всіх операцій, практичні застосування та методи реалізації черги з пріоритетами. Особливу увагу приділимо візуалізації операцій та покроковому аналізу алгоритмів.

Теоретичні основи бінарної купи

Означення та властивості

Бінарна купа - це повне бінарне дерево, що задовольняє властивість купи. Структура поєднує дві фундаментальні характеристики:

Структурна властивість (Shape Property): Бінарна купа є повним бінарним деревом - усі рівні заповнені повністю, за винятком можливо останнього

рівня. Останній рівень заповнюється зліва направо без пропусків. Ця властивість забезпечує збалансованість дерева та логарифмічну висоту.

Властивість порядку (Heap Property):

- Максимальна купа (Max-Heap): кожен батьківський вузол більший або дорівнює своїм дочірнім вузлам: A[parent(i)] ⩾ A[i]

- Мінімальна купа (Min-Heap): кожен батьківський вузол менший або дорівнює своїм дочірнім вузлам: A[parent(i)] ⩽ A[i]

Корінь містить екстремальний елемент: у максимальній купі корінь містить найбільший елемент, у мінімальній купі - найменший. Це робить операцію отримання мінімального або максимального значення надзвичайно ефективною.

Часткова упорядкованість: на відміну від бінарного дерева пошуку, у бінарній купі відсутня строга упорядкованість між сусідніми вузлами одного рівня. Це спрощує алгоритми вставлення та видалення.

Бінарна купа ефективно представляється у вигляді масиву, використовуючи обхід дерева в ширину. Це значно спрощує реалізацію та покращує локальність пам’яті.

Нагадаємо, що основною характеристикою дерева є висота.

Для повного бінарного дерева з n вузлами висота h обчислюється як:

$h = \lfloor \log_2(n) \rfloor$

Доведення: Повне бінарне дерево висоти h має від 2^h до 2^(h+1) - 1 вузлів. Тому:

$2^h \leq n \leq 2^{(h+1)} - 1$

Застосовуючи логарифм за основою 2:

$h \leq \log_2(n) < h + 1$

Звідси: $h = \lfloor \log_2(n) \rfloor$

8.2.2 Кількість вузлів на кожному рівні

- Рівень 0 (корінь): 1 вузол
- Рівень i: максимум 2^i вузлів
- Загальна кількість вузлів у повному дереві висоти h:

$сума(i=0\ to\ h)\ 2^i = 2^{(h+1)} - 1$

Для реалізації купи за допомогою (починаючи з індексу 1) мажмо такі формули для індексів елементів масиву, в який зберігається **i-й** вузол:

- Батьківський вузол для вузла i: ⌊i/2⌋

- Лівий нащадок вузла i: 2i

- Правий нащадок вузла i: 2i + 1

- Листкові вузли: індекси від ⌊n/2⌋ + 1 до n

Для альтернативної індексації (починаючи з індексу 0):

- Батьківський вузол для вузла i: ⌊(i-1)/2⌋

- Лівий нащадок вузла i: 2i + 1

- Правий нащадок вузла i: 2i + 2

Розглянемо операції над бінарною купою .

Операція вставки

Алгоритм вставки:

1. Додати новий елемент у наступну доступну позицію (в кінеці масиву)

2. Відновити властивість купи шляхом “спливання вгору”

Псевдокод:

```
ВСТАВИТИ(купа, ключ):
    купа.розмір = купа.розмір + 1
    і = купа.розмір
    поки і > 1 та купа[БАТЬКО(і)] < ключ:  // для максимальної купи
        купа[і] = купа[БАТЬКО(і)]
        і = БАТЬКО(і)
    купа[і] = ключ
```

Складність часу: O(log n) - у найгіршому випадку потрібно пройти від листка до кореня.

Покроковий приклад вставлення:

Вставимо 15 у максимальну купу [20, 18, 10, 12, 9, 5, 6, 7, 11]

1. Початковий стан: [20, 18, 10, 12, 9, 5, 6, 7, 11]
2. Крок 1: Додаємо 15 у позицію 10: [20, 18, 10, 12, 9, 5, 6, 7, 11, 15]
3. Крок 2: Порівнюємо з батьком (9): 15 > 9, міняємо місцями
4. Крок 3: Порівнюємо з новим батьком (18): 15 < 18, зупиняємося
5. Фінальний стан: [20, 18, 10, 12, 15, 5, 6, 7, 11, 9]

Операція видалення максимуму/мінімуму

Алгоритм видалення максимуму:

1. Зберегти значення кореня (екстремальний елемент)

2. Замінити корінь останнім елементом

3. Видалити останній елемент

4. Відновити властивість купи шляхом “занурення вниз”

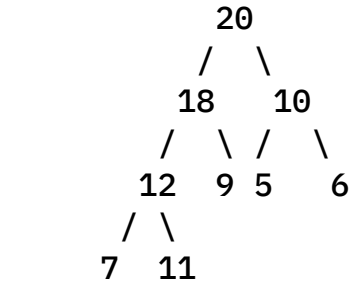
Псевдокод:

```
ВИДАЛИТИ_МАКСИМУМ(купа):
    якщо купа.розмір < 1:
        помилка "купа порожня"
    максимум = купа[1]
    купа[1] = купа[купа.розмір]
    купа.розмір = купа.розмір - 1
    МАКСИМАЛЬНА_КУПІФІКАЦІЯ(купа, 1)
    повернути максимум
```

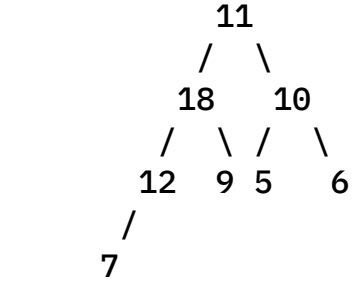
Складність часу: O(log n)

Покроковий приклад видалення максимуму.

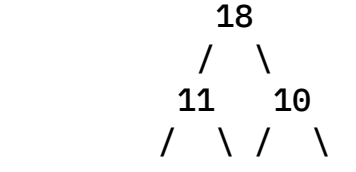
Початкова максимальна купа: [20, 18, 10, 12, 9, 5, 6, 7, 11]



Крок 1: Зберігаємо максимум (20) Крок 2: Переміщуємо останній елемент (11) в корінь



Крок 3: Порівнюємо 11 з дочірніми вузлами (18, 10) - 18 більший Крок 4: Міняємо місцями 11 та 18



```
    12  9 5   6
   /
  7
```

Крок 5: Порівнюємо 11 з дочірніми вузлами (12, 9) - 12 більший Крок 6: Міняємо місцями 11 та 12

```
        18
       /  \
      12   10
     /  \ /  \
    11  9 5   6
   /
  7
```

Крок 7: 11 має лише одного дочірнього вузла (7), 11 > 7, зупиняємося Фінальний результат: [18, 12, 10, 11, 9, 5, 6, 7]

Операція перетворення звичайного масиву на структуру купи (Heapify, купіфікація**)**

Максимальна купіфікація (занурення вниз):

```
МАКСИМАЛЬНА_КУПІФІКАЦІЯ(купа, і):
    л = ЛІВИЙ(і)
    п = ПРАВИЙ(і)
    якщо л ≤ купа.розмір та купа[л] > купа[і]:
        найбільший = л
    інакше:
        найбільший = і
    якщо п ≤ купа.розмір та купа[п] > купа[найбільший]:
        найбільший = п
    якщо найбільший ≠ і:
        поміняти_місцями купа[і] з купа[найбільший]
        МАКСИМАЛЬНА_КУПІФІКАЦІЯ(купа, найбільший)
```

Складність часу: O(log n)

Побудова купи - метод Флойда

Алгоритм побудови купи:

1. Почати з довільного масиву

2. Застосувати купіфікацію від останнього не-листкового вузла до кореня

Псевдокод:

```
ПОБУДУВАТИ_МАКСИМАЛЬНУ_КУПУ(масив):
    купа.розмір = масив.довжина
    для і = ⌊купа.розмір/2⌋ вниз_до 1:
```

```
МАКСИМАЛЬНА_КУПІФІКАЦІЯ(купа, і)
```

Складність часу: O(n) - точний аналіз показує лінійну складність, а не O(n log n)

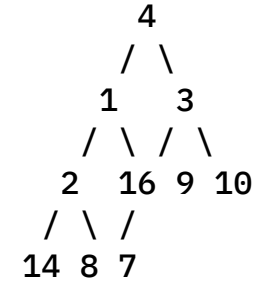
Математичне доведення O(n) складності:

- Висота купи = ⌊log n⌋
- Кількість вузлів на висоті h: щонайбільше ⌈n/2^(h+1)⌉
- Робота на висоті h: O(h) на вузол
- Загальна робота: Σ(h=0 to ⌊log n⌋) ⌈n/2^(h+1)⌉ * O(h)
- Це дорівнює O(n * Σ(h=0 to ∞) h/2^h) = O(n * 2) = O(n)

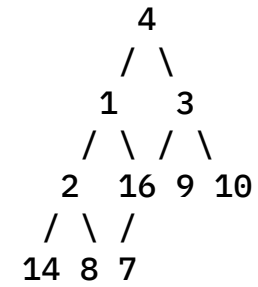
Візуалізація побудови купи

Початковий масив: [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

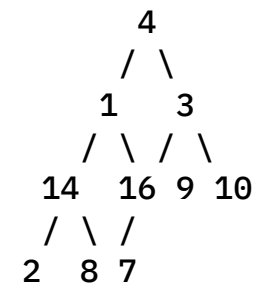
Крок 1: Початкове дерево (не купа)



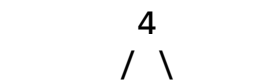
Крок 2: Застосовуємо купіфікацію до вузла 5 (значення 16)

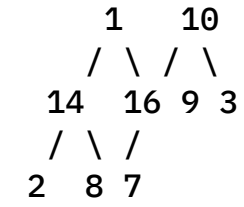


Крок 3: Застосовуємо купіфікацію до вузла 4 (значення 2)

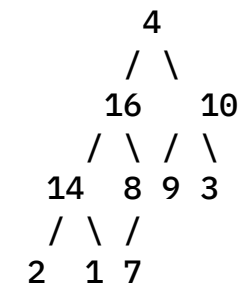


Крок 4: Застосовуємо купіфікацію до вузла 3 (значення 3)

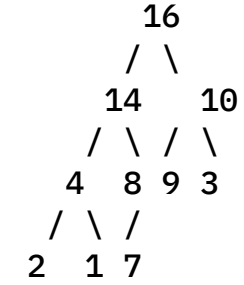




Крок 5: Застосовуємо купіфікацію до вузла 2 (значення 1)



Крок 6: Застосовуємо купіфікацію до вузла 1 (значення 4)



Фінальна максимальна купа: [16, 14, 10, 4, 8, 9, 3, 2, 1, 7]

Додаткові операції

Зменшення ключа (для мінімальної купи):

Операція зменшення ключа у мін-купі потрібна для підтримки інваріанту та коректної роботи пріоритетної черги у випадках, коли пріоритет (ключ) певного елемента з часом стає вищим (меншим числом). Розберемо детально.

У багатьох алгоритмах та застосунках нам не достатньо одноразово вставити елемент із фіксованим пріоритетом — його пріоритет може оновлюватися в процесі виконання:

- Графові алгоритми (Дейкстри, A*): після першої вставки вершини в чергу ми дізнаємося кращу (коротшу) відстань до неї й мусимо зменшити її ключ у черзі.
- Симуляції подій: час події може бути скоригований (відкласти раніше заплановану подію).

– Планувальники процесів: пріоритет задачі може змінитися через підвищення її важливості.

Без операції decrease_key нам довелося б:

1. Шукати цей елемент у масиві купи (O(n)), видаляти його, а потім вставляти з новим ключем (загалом O(n) + O(log n)),

   або

2. Зберігати додаткові структури для позначення «старої» версії елемента й ігнорувати її під час видобування, що ускладнює реалізацію й може призвести до накопичення «сміття».

У мін-купі інваріант такий:

для будь-якого вузла i справджується A[parent(i)] ⩽ A[i].

Якщо ми зменшуємо ключ елемента в позиції i (тобто робимо новий A[i] < старий A[i]), порушення може з’явитися лише між цим вузлом і його батьком (оскільки всі інші пари зв’язків не змінювалися).

Щоб відновити інваріант, достатньо локально просіяти цей елемент вгору (sift_up), порівнюючи з батьком та обмінюючи в разі потреби, доки A[parent] ⩽ A[current].

Оскільки висота купи ≈ $\lfloor \log_2 n \rfloor$, операція займає O(log n).

Алгоритм decrease_key

1. Перевірка

   Якщо `new_key > A[i]`, це не зменшення — повідомляємо помилку.

2. Присвоєння

   `A[i] ← new_key`.

3. Локальна корекція

   Повторюємо, поки i > 1 і `A[i] < A[parent(i)]`:

   1. Міняємо `A[i]` і `A[parent(i)]`.
   2. i ← parent(i).

Після завершення гарантовано відновлено інваріант у всьому дереві, бо єдиний можливий шлях порушення — від початкової позиції і догори.

Збільшення ключа (для максимальної купи):

Операція increase_key у макс-купі виконує ту саму роль, що й decrease_key у мін-купі — вона дозволяє ефективно змінювати пріоритет вже існуючого елемента без повного видалення та повторної вставки. Розгорнемо ідею детально:

У багатьох задачах пріоритет елемента може збільшуватися в процесі роботи програми, наприклад:

- Планувальники ресурсів: процес отримує вищий пріоритет через критичні строки виконання.
- Онлайн-ігри чи системи реального часу: користувач чи об'єкт раптово стає «важливішим».
- Системи термінових сповіщень: повідомлення можуть раптово підвищити рівень важливості.

Якщо б ми просто змінили поле `key` у масиві купи, інваріант макс-купи (батько ⩾ діти) міг би порушитися лише на шляху від цієї вершини до низу — тобто між новим ключем і його дітьми.

У макс-купі інваріант звучить так:

для будь-якого вузла i, якщо у нього є діти, то `A[parent(i)] ≥ A[i].`

Коли ключ у позиції i збільшився (новий `A[i]` > старий), порушення може виникнути лише на відношенні `A[i]` з його дітьми, бо раніше інваріант тримався і між усіма іншими парами зв'язків. Щоб відновити інваріант, потрібно просіяти цей елемент вниз (sift_down), обмінюючи його з тією дитиною, яка має більший ключ, доки обидва нащадки не будуть меншого або рівного значення.

Кроки операції increase_key такі:

1. Перевірка коректності

   Якщо `new_key < A[i]`, це не збільшення пріоритету — помилка.

2. Присвоєння нового ключа

   `A[i] ← new_key`.

3. Локальна корекція вниз

```
while true
    l ← 2*i,  r ← 2*i + 1
    best ← i
    if l ≤ heap_size and A[l] > A[best]
        best ← l
    if r ≤ heap_size and A[r] > A[best]
        best ← r
    if best = i
        break    # інваріант відновлено
    swap A[i] with A[best]
    i ← best
end while
```

- Обираємо ту дитину, в якої найбільший ключ, і міняємо місцями.
- Продовжуємо, доки позиція і стає «вищою» за обидві дитини.

Просіювання вниз проходить не більше ніж по висоті дерева: O(log n).

Якщо б у нас не було `increase_key`, то при зміні пріоритету довелося б:

1. Знайти у масиві позицію i (O(1), якщо ми відразу знаємо індекс, або O(n), якщо потребуємо пошуку).
2. Видалити елемент (через заміну останнім + sift_down/up) — O(log n).
3. Вставити його з новим ключем — ще O(log n).

Іншими словами, без прямої операції було б мінімум 2 × O(log n) або навіть O(n) + O(log n), замість одного O(log n).

Видалення довільного елемента:

Видалення не лише екстремального елемента (кореня), а будь-якого вузла i — неодмінна операція, коли нам потрібно прибрати з пріоритетної черги

якийсь конкретний елемент (наприклад, скасувати задачу за індексом або id).

У пріоритетній черзі іноді потрібно не тільки «взяти найменший» чи «найбільший», а видалити конкретний елемент, наприклад:

- Скасувати завдання, що ще чекає в черзі, за його ідентифікатором.
- Видалити запис у симуляції подій, якщо подія стала недійсною.
- Очистити кеш, прибравши з купи конкретний запис.

Без цієї операції довелося б реалізовувати складні обходи чи маркування «видалених» елементів, що ускладнює логіку.

Розглянемо принцип роботи операції видалення.

1. Заміна на кінець
    - Щоб зберегти форму повного дерева, спочатку беремо останній елемент масиву (`A[heap_size]`) і переміщаємо його в позицію i, яку потрібно видалити.
    - Зменшуємо `heap_size` на 1 — останній елемент, який вилучено.
2. Двонаправлена корекція інваріанту
    - Після заміщення у позиції i може порушитися інваріант купи:
        - Якщо новий `A[i]` менший/кращий за батька (для мін-купи) → інваріант порушено угору.
        - Якщо новий `A[i]` гірший за якийсь з дітей → інваріант порушено донизу.
    - Тому одночасно перевіряємо обидва напрями:
        1. Якщо `i > 1` і новий ключ кращий за ключ батька → застосовуємо sift_up.
        2. Інакше → застосовуємо sift_down.

Ми коригуємо інваріант лише вздовж одного шляху (вгору чи вниз), не зачіпаючи інших гілок. Кожна з операцій — O(log n), бо висота купи логарифмічна.

Наведемо псевдокод операції

```
procedure heap_delete(A, i)
    if i < 1 or i > heap_size
        error "Index out of range"

     1) Заміна
    A[i] ← A[heap_size]
    heap_size ← heap_size - 1

     2) Визначаємо напрям корекції
    if i > 1 and A[i] < A[floor(i/2)] мін-купа, перевіряємо з батьком
        sift_up(A, i)                 піднімаємо новий елемент
    else
        sift_down(A, i, heap_size)    опускаємо його, якщо треба
    end if
end procedure
```

Примітка: для макс-купи нерівності, звісно, протилежні:

– Перевірка з батьком: `A[i] > A[parent]` → `sift_up`.

– Для дітей: обираємо ту дитину, де ключ більший.

7. Розглянемо виконання операції на прикладі.

Нехай маємо мін-купу, масив і `heap_size = 7`:

```
Індекс:  1   2   3   4   5   6   7
Масив:  [2,  5,  7, 10, 12, 8,  15]
```

Потрібно видалити елемент за індексом 2 (ключ = 5, значення умовне).

1. Заміна: копіюємо останній елемент (15) в позицію 2 і зменшуємо розмір:

   `[2, 15, 7, 10, 12, 8]   (heap_size = 6)`

2. Перевірка з батьком: батько позиції 2 — індекс 1, ключ = 2.

   - $15 \geqslant 2 \Rightarrow$ порушення інваріанту угору немає.

3. Просіювання вниз (`sift_down(2)`):

- Діти позиції 2: індекси 4 (10) і 5 (12).
- Мінімальна дитина — індекс 4, ключ 10.
- 15 > 10 ⇒ інваріант порушено, міняємо:

  `[2, 10, 7, 15, 12, 8]`
- Нова позиція i=4; діти індекси 8 і 9 виходять за `heap_size` ⇒ зупинка.

В результаті видалення індекс 2, масив залишається коректною мін-купою .

Аналіз складності.

– Часова: O(log n) у найгіршому випадку (один виклик sift_up або sift_down).

– Середня часова складність: O(log n)

– Пам’ять: O(1) додатково, лише кілька змінних й обмінів.

Наведемо типові помилки при реалізації операцій:

1. Не перевіряють індекс: і може бути за межами [1…heap_size].
2. Завжди викликають `sift_down` навіть коли новий ключ кращий за батька, і навпаки.
3. Неправильно обчислюють parent(i) (наприклад, не використовують цілочисельне ділення).
4. Не зменшують heap_size до корекції, що може призвести до порушення коректності при `sift_down`.

Реалізація операції **Перегляд максимуму/мінімуму** дуже проста. Наведемо її псевдокод:

```
ПЕРЕГЛЯНУТИ_МАКСИМУМ(купа):
    якщо купа.розмір < 1:
        помилка "купа порожня"
    повернути купа[1]
```

Складність часу: O(1)

Обхід бінарної купи

Обходи бінарного дерева — базові процедури, які послідовно відвідують усі вузли за певним правилом. Хоч двійкова купа не призначена для пошуку за ключем, усі чотири стандартні обходи (префіксний, інфіксний, постфіксний і рівневий) важливі для ознайомлення зі структурою дерева та для загального розуміння механіки рекурсії і черги.

Усі наведенні алгоритми працюють із масивною репрезентацією купи `A[1..heap_size]`, де для індексу `i`:

- `left(i) = 2·i` — лівий нащадок (якщо `2·i ≤ heap_size`),
- `right(i) = 2·i + 1` — правий (якщо `2·i + 1 ≤ heap_size`).

Префіксний (preorder)

Правило:

Відвідати вузол → обійти ліве піддерево → обійти праве піддерево.

Логіка й мотивація

- Відвідати вузол одразу, перш ніж заходити в будь-які нащадки.
- Корисно, коли треба скопіювати структуру дерева «зверху вниз» або виконати операцію над кожним вузлом до обробки дітей.

Псевдокод

```
procedure preorder(i)
    if i > heap_size
        return
    visit(A[i])                 дія над поточним вузлом
    preorder(2 * i)             рекурсивно ліве піддерево
    preorder(2 * i + 1)         рекурсивно праве піддерево
end procedure
```

Складність

- Час: O(n) — кожен вузол відвідується рівно один раз.
- Пам’ять: O(h) стек викликів рекурсії, де $h = \lfloor \log_2 n \rfloor$.

Інфіксний (inorder)

Правило:

Обійти ліве піддерево → відвідати вузол → обійти праве піддерево.

Логіка й мотивація

– У бінарному пошуковому дереві (не купі) дає зростаючий порядок значень.

– Для купи цей порядок не впорядковує, але демонструє механізм «відкладеного» відвідання вузла між піддеревами.

Псевдокод

```
procedure inorder(i)
    if i > heap_size
        return
    inorder(2 * i)              ліве піддерево
    visit(A[i])                 поточний вузол
    inorder(2 * i + 1)          праве піддерево
end procedure
```

Особливості

– На купі результат буде «невпорядкований» щодо ключів, але алгоритм ілюструє принцип рекурсії «знизу вгору».

Постфіксний (postorder)

Правило:

Обійти ліве піддерево → обійти праве піддерево → відвідати вузол.

Логіка й мотивація

– Використовують, коли спочатку потрібно обробити/звільнити всі нащадки, а потім сам вузол (наприклад, звільнення пам’яті).

– «Збирає» дані з доньок перед батьком.

Псевдокод

```
procedure postorder(i)
    if i > heap_size
        return
    postorder(2 * i)            ліве піддерево
    postorder(2 * i + 1)        праве піддерево
    visit(A[i])                 поточний вузол
end procedure
```

Рівневий (level-order)

Правило:

Відвідати вузли по рівнях зверху вниз, зліва направо.

Логіка й мотивація

- Сам по собі рівневий обхід збігається з порядком зберігання купи в масиві.
- Зручно, коли треба вивести «зрізи» дерева або перевірити структуру повноти.

Псевдокод (циклом)

```
procedure level_order()
    for i from 1 to heap_size
        visit(A[i])
    end for
end procedure
```

Порівняльний аналіз обходів

Обхід	Порядок відвідування	Використання
preorder	вузол → ліво → право	копіювання/клонування дерева
inorder	ліво → вузол → право	в BST — отримати відсортовану послідовність
postorder	ліво → право → вузол	звільнення/видалення, агрегування знизу
level-order	рівні зверху вниз, зліва направо	перевірка повноти, друк «зрізів» дерева

Абстрактний тип даних "Черга з пріоритетами"

Необхідність використання черги з пріорітетами ми вже розглянули. Тепер перейдемо до реалізації цієї структури даних.

Операції черги з пріоритетами:

- вставити(елемент, пріоритет): додати елемент із заданим пріоритетом

- видалити_мінімум()/видалити_максимум(): видалити та повернути елемент з найвищим пріоритетом

- знайти_мінімум()/знайти_максимум(): повернути елемент з найвищим пріоритетом без видалення

- порожня(): перевірити, чи черга порожня - розмір(): повернути кількість елементів

```
Клас ПріоритетнаЧерга
    // Масивове представлення купи: індекс 1…розмір
    поле купа[ ]
    поле розмір

    // Конструктор
    процедура Ініціалізувати()
        купа ← масив із одним елементом (порожній)  // індекс 0 не ви
користовується
        розмір ← 0
    кінець процедури

    // Вставити елемент із пріоритетом
    процедура Вставити(елемент)
        розмір ← розмір + 1
        купа[розмір] ← елемент
        СплівтиВгору(розмір)
    кінець процедури

    // Видалити та повернути мінімальний елемент
    процедура ВидалитиМінімум() → елемент
        якщо розмір = 0
            помилка "Порожня черга"
        кінь_значення ← купа[1]
        // Переносимо останній елемент у корінь
        купа[1] ← купа[розмір]
        розмір ← розмір – 1
        видалити останній елемент із купи
        якщо розмір > 0
            ЗануритиВниз(1)
        повернути кінь_значення
    кінець процедури

    // Локальна корекція вгору після вставки
    процедура СплівтиВгору(i)
        поки i > 1 і купа[i] < купа[floor(i / 2)]
            поміняти місцями купа[i] і купа[floor(i / 2)]
            i ← floor(i / 2)
        кінець поки
    кінець процедури

    // Локальна корекція вниз після видалення кореня
    процедура ЗануритиВниз(i)
        поки 2·i ≤ розмір
            j ← НайменшийДитина(i)
```

```
            якщо купа[і] > купа[j]
                поміняти місцями купа[і] і купа[j]
            кінець якщо
            і ← j
        кінець поки
    кінець процедури

    // Допоміжна: знайти індекс меншої дитини
    процедура НайменшийДитина(і) → індекс
        ліва ← 2·і
        права ← 2·і + 1
        якщо права > розмір
            повернути ліва
        інакше
            якщо купа[ліва] < купа[права]
                повернути ліва
            інакше
                повернути права
            кінець якщо
        кінець якщо
    кінець процедури
кінець класу
```

Пояснення:

- Вставити додає елемент у кінець і піднімає його вгору до відновлення інваріанту.
- ВидалитиМінімум міняє корінь на останній елемент, зменшує розмір і «занурює» корінь вниз.
- СплівтиВгору та ЗануритиВниз коригують інваріант купи за допомогою локальних обмінів.
- НайменшийДитина обирає між двома дітьми ту, що менша, для мін-купи.

Застосування та випадки використання

- Планування задач в операційних системах: Бінарна купа використовується для керування чергами процесів з різними пріоритетами. Операційна система може ефективно вибирати наступний процес для виконання, базуючись на пріоритеті.
- Алгоритми на графах: - Алгоритм Дейкстри: знаходження найкоротших шляхів у графі - Алгоритм Пріма: побудова

мінімального остовного дерева - Алгоритм A*: пошук шляхів з евристикою

- Кодування Гафмана: стиснення даних з використанням частотних характеристик символів. Купа використовується для побудови оптимального дерева кодування.
- Подійно-керована симуляція: керування подіями за часом їх виконання. Найближча за часом подія завжди знаходиться в корені купи.

Спеціальні випадки використання

- Управління відділенням швидкої допомоги: Пріоритизація пацієнтів за тяжкістю стану. Найтяжчі випадки завжди обслуговуються першими.
- Планування завдань процесора: Керування завданнями з різними пріоритетами в багатозадачних системах.
- Планування рейсів: Керування пріоритетами зльоту та посадки літаків в аеропортах.
- Керування чергою друку: Обробка завдань друку з урахуванням пріоритетів користувачів.
- Торгові системи: Обробка замовлень на біржі за пріоритетом та часом надходження.

Переваги та недоліки

Переваги

- Ефективність за часом: O(log n) для операцій вставлення та видалення забезпечує відмінну продуктивність навіть для великих наборів даних.
- Ефективність за пам’яттю: Масивна реалізація має мінімальні накладні витрати пам’яті порівняно з деревоподібними структурами з вказівниками.
- Простота реалізації: Прямолінійна масивна структура спрощує написання та налагодження коду.
- Оптимізація кешу: Масивний доступ покращує локальність пам’яті та продуктивність кешу процесора.
- Операції на місці: Більшість операцій не потребують додаткової пам’яті.

Недоліки

- Відсутність пошуку: Знаходження довільних елементів потребує O(n) часу, що неефективно для великих структур.
- Неповна упорядкованість: Елементи не знаходяться в повністю відсортованому порядку.
- Складність зміни пріоритетів: Зміна пріоритетів елементів потребує знання їхньої позиції в купі.
- Відсутність операції злиття: Об’єднання двох куп є дорогою операцією O(n).

Контрольні питання

1. Яка основна різниця між структурною властивістю та властивістю порядку в бінарній купі?
2. Чому висота повного бінарного дерева з n вузлами дорівнює $\lfloor \log_2(n) \rfloor$?
3. Як знайти індекс батьківського вузла для елемента в позиції i при масивній реалізації купи?
4. Чому операція побудови купи має складність O(n), а не O(n log n)?
5. В яких випадках доцільно використовувати мінімальну купу замість максимальної?
6. Як відрізняється алгоритм видалення кореня від алгоритму видалення довільного елемента?
7. Чому бінарна купа не підходить для швидкого пошуку довільних елементів?
8. Як масивна реалізація купи впливає на продуктивність порівняно з деревоподібною реалізацією?
9. Які алгоритми на графах використовують бінарну купу і чому?
10. Як можна модифікувати бінарну купу для підтримки операції зменшення ключа?

Методичні рекомендації до виконання лабораторної роботи

1 Мета роботи

Реалізувати бінарну купу та чергу з пріоритетами, продемонструвати роботу основних операцій, дослідити ефективність алгоритмів.

2 Завдання

Завдання 1: Реалізувати клас **БінарнаКупа** з наступними методами: - **вставити(елемент)** - додати елемент - **видалити_максимум()** - видалити та повернути максимальний елемент - **переглянути_максимум()** - повернути максимальний елемент - **порожня()** - перевірити, чи купа порожня - **розмір()** - повернути кількість елементів

Завдання 2: Реалізувати метод **побудувати_купу(масив)** для побудови купи з довільного масиву.

Завдання 3: Додати методи для роботи з пріоритетами: - **вставити_з_пріоритетом(елемент, пріоритет)** - **зменшити_пріоритет(індекс, новий_пріоритет)** - **видалити_елемент(індекс)**

Завдання 4: Реалізувати алгоритм сортування купою (HeapSort).

Завдання 5: Дослідити ефективність операцій шляхом вимірювання часу виконання для різних розмірів даних.

Тестові дані

```
# Тестові масиви
тест1 = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
тест2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  # вже відсортований
тест3 = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]  # обернено відсортований
тест4 = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5]   # однакові елементи
тест5 = [42]  # один елемент
тест6 = []    # порожній масив
```

4 Очікувані результати

Після виконання лабораторної роботи студенти повинні продемонструвати: - Правильну реалізацію всіх операцій купи - Розуміння різниці між мінімальною та максимальною купою - Вміння аналізувати складність алгоритмів - Практичне застосування купи для розв’язання задач

5 Звіт

Звіт повинен містити:

1. Код реалізації з коментарями

2. Результати тестування на всіх тестових даних

3. Аналіз складності кожної операції

4. Графіки ефективності для різних розмірів даних

5. Висновки про переваги та недоліки реалізації

Тести

1. Яка з наведених властивостей НЕ є обов’язковою для бінарної купи?

a) Повне бінарне дерево

b) Властивість порядку (heap property)

c) Всі листки знаходяться на одному рівні

d) Кожен батьківський вузол має щонайбільше двох дочірніх вузлів

2. Для максимальної купи з масивом [16, 14, 10, 8, 7, 9, 3, 2, 4, 1], який елемент знаходиться в позиції 3?

a) 16

b) 14

c) 10

d) 8

3. Яка складність операції вставлення елемента в бінарну купу?

a) O(1)

b) O(log n)

c) O(n)

d) O(n log n)

4. Скільки порівнянь потрібно в найгіршому випадку для видалення кореня з купи розміру 15?

a) 3

b) 4

c) 7

d) 8

5. Яка з наведених формул правильно обчислює індекс батьківського вузла для елемента в позиції i (індексація з 1)?

a) i / 2

b) ⌊i / 2⌋

c) i * 2

d) i - 1

Відповіді. 1. - c, 2. - c , 3. - b, 4. - a, 5. - b

Тестові завдання підвищеної складності

1. Для побудови максимальної купи з масиву [3, 5, 1, 10, 4, 6, 8] методом Флойда, яка буде послідовність застосування операції heapify?

a) Індекси 3, 2, 1

b) Індекси 4, 3, 2, 1

c) Індекси 3, 2, 1, 0

d) Індекси 7, 6, 5, 4, 3, 2, 1

2. Якщо в мінімальній купі потрібно зменшити значення елемента в позиції 8, який алгоритм буде використаний?

a) Перколяція вниз (percolate down)

b) Перколяція вгору (percolate up)

c) Повна перебудова купи

d) Видалення та повторне вставлення

3. Яка мінімальна кількість обмінів потрібна для перетворення масиву [1, 2, 3, 4, 5, 6, 7] у максимальну купу?

a) 0

b) 2

c) 3

d) 4

4. В алгоритмі Дейкстри для графа з V вершинами та E ребрами, яка загальна складність при використанні бінарної купи?

a) O(V log V)

b) O(E log V)

c) O((V + E) log V)

d) O(V + E log V)

5. Яка максимальна кількість елементів може бути в двійковій купі висоти 4?

a) 15

b) 16

c) 31

d) 32

6. Для операції злиття двох двійкових куп розміру n кожна, яка найкраща можлива складність?

a) O(log n)

b) O(n)

c) O(n log n)

d) $O(n^2)$

7. Які дві операції разом еквівалентні операції decrease-key в мінімальній купі?

a) Insert + Extract-Min

b) Delete + Insert

c) Extract-Min + Insert

d) Build-Heap + Extract-Min

8. Яка складність пошуку елемента зі значенням x в довільній бінарній купі?

a) O(1)

b) O(log n)

c) O(n)

d) O(n log n)

Відповіді.

1. a : Починаємо з останнього не-листкового вузла ⌊7/2⌋ = 3.

2. b : Після зменшення значення елемент може порушувати властивість купи знизу.

3. c : Потрібно перемістити елементи для формування максимальної купи.

4. c : V операцій extract-min плюс E операцій decrease-key.

5. c : Повне бінарне дерево висоти 4 має $2^5 - 1 = 31$ вузол.

6. b : Найкращий алгоритм злиття має лінійну складність.

7. b : Видалити елемент і вставити його з новим значенням.

8. c : Купа не підтримує ефективний пошук довільних елементів.

Завдання для самостійної роботи

1. Доведіть, що побудова купи методом Флойда має складність O(n). Наведіть детальний математичний аналіз.
2. Порівняйте ефективність трьох способів побудови купи з n елементів:
    - Послідовне вставлення n елементів
    - Метод Флойда

- Сортування з наступним переупорядкуванням

3. Дослідіть d-арну купу (купу з d дочірніми вузлами). Виведіть формули для знаходження батьківського вузла та дочірніх вузлів.

5. Реалізуйте алгоритм сортування купою та порівняйте його ефективність із швидким сортуванням та сортуванням злиттям.

6. Реалізуйте структуру даних для підтримки медіани потоку чисел, використовуючи дві купи.

7. Розробіть алгоритм для знаходження k найбільших елементів у масиві розміру n з оптимальною складністю.

7. Використайте мінімальну купу для ефективного злиття k відсортованих списків.

8. Реалізуйте симуляцію планувальника завдань операційної системи з використанням купи з пріоритетами.

9. Реалізуйте алгоритм Дейкстри для знаходження найкоротших шляхів у графі з використанням бінарної купи.

10. Проведіть експериментальне дослідження ефективності різних реалізацій черги з пріоритетами (бінарна купа, AVL-дерево, несортований масив) для різних розмірів даних та типів операцій.

Біноміальні купи

1. Теоретичні відомості

У світі алгоритмів та структур даних, купи є фундаментальним інструментом для ефективного управління пріоритетами. Вони дозволяють швидко знаходити мінімальний (або максимальний) елемент, вставляти нові елементи та видаляти існуючі. Проте, звичайні бінарні купи, хоча й прості у реалізації, мають певні обмеження щодо ефективності деяких операцій, особливо при об’єднанні двох куп. Саме тут на сцену виходять біноміальні купи – більш складна, але надзвичайно потужна структура даних, яка забезпечує кращу амортизовану продуктивність для широкого спектру операцій, зокрема для операції об’єднання.

Біноміальна купа – це не просто одне дерево, а колекція (або ліс) спеціальних дерев, які називаються біноміальними деревами. Кожне

біноміальне дерево в цій колекції має унікальні властивості, що робить біноміальну купу такою ефективною.

1.1. Біноміальні дерева (B_k)

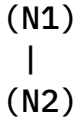
Перш ніж заглиблюватися в біноміальні купи, необхідно зрозуміти їхні будівельні блоки – біноміальні дерева. Біноміальне дерево порядку k, позначається як B_k, є рекурсивно визначеним деревом. Позначимо вузол дерева буквою в дужках, наприклад: (N)

1. Біноміальне дерево порядку 0 (B_0): Це одновузлове дерево. Воно складається лише з одного вузла.

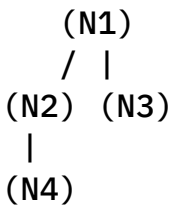
(N)

2. Біноміальне дерево порядку k (B_k): Формується шляхом об'єднання двох біноміальних дерев порядку $k-1$. Одне з дерев стає лівим дочірнім елементом кореня іншого дерева. Важливо, що корінь одного B_k-1 стає дочірнім елементом кореня іншого B_k-1. Зазвичай, дерево, яке стає дочірнім, має корінь з більшим значенням, щоб зберегти властивість купи.

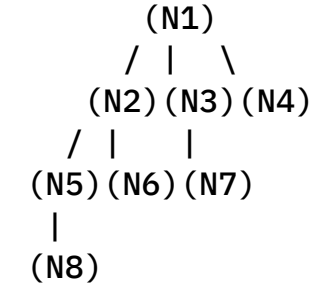
- B_1: Утворюється шляхом об'єднання двох B_0. Один B_0 стає дочірнім елементом іншого B_0.



- B_2: Утворюється шляхом об'єднання двох B_1. Корінь одного B_1 стає дочірнім елементом кореня іншого B_1.



- B_3: Утворюється шляхом об'єднання двох B_2.



Властивості біноміального дерева B_k:

1. Кількість вузлів: Біноміальне дерево B_k має точно $2k$ вузлів.
    - Для B_0: 2^0=1 вузол.
    - Для B_1: 2^1=2 вузли.
    - Для B_2: 2^2=4 вузли.
    - Для B_3: 2^3=8 вузлів.
    - Це випливає з рекурсивного визначення: B_k складається з двох $B_k - 1$, тому кількість вузлів подвоюється на кожному кроці.
2. Висота: Висота біноміального дерева B_k дорівнює k. Висота дерева – це довжина найдовшого шляху від кореня до будь-якого листа.
    - B_0: висота 0.
    - B_1: висота 1.
    - B_2: висота 2.
    - B_3: висота 3.
3. Кількість вузлів на глибині i: На глибині i (де корінь знаходиться на глибині 0), біноміальне дерево B_k має C_k^i вузлів. Це є причиною назви “біноміальне”, оскільки кількість вузлів на кожній глибині відповідає біноміальним коефіцієнтам.

    Наприклад, для B_3:
    - Глибина 0: C_3^0=1 вузол (корінь).
    - Глибина 1: C_3^1=3 вузли.
    - Глибина 2: C_3^2=3 вузли.
    - Глибина 3: C_3^1=1 вузол.
    -
4. Ступінь кореня: Корінь біноміального дерева B_k має ступінь k. Ступінь вузла – це кількість його дочірніх елементів. Дочірні елементи кореня B_k є коренями біноміальних дерев $B_k - 1, B_k -$

2,…,B_0. Це важлива властивість, яка використовується при об'єднанні дерев.

1.2. Структура Біноміальної купи

Біноміальна купа H – це колекція (або ліс) біноміальних дерев, яка задовольняє двом ключовим властивостям:

1. Властивість купи: Кожне біноміальне дерево в купі задовольняє властивості мінімальної купи: ключ кожного вузла більший або дорівнює ключу його батьківського вузла. Це означає, що найменший ключ у кожному біноміальному дереві знаходиться в його корені.
2. Унікальність порядку: Для будь-якого цілого числа $k \geq 0$, у біноміальній купі може бути щонайбільше одне біноміальне дерево порядку k. Це означає, що якщо ми маємо біноміальні дерева $B_0, B_1, B_2, \ldots, B_m$, то кожне з цих порядків може зустрічатися в купі не більше одного разу.

Ці дві властивості разом гарантують, що біноміальна купа з n вузлами має унікальну структуру. Кількість вузлів n може бути виражена в двійковій системі числення як

$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i \, 2^i$ де $b_i \in (0,1)$.

Якщо $b_i = 1$, то в біноміальній купі присутнє біноміальне дерево B_i. Таким чином, біноміальна купа з n вузлами складається з $\lfloor \log_2 n \rfloor + 1$ біноміальних дерев, що відповідають одиницям у двійковому представленні n.

Наприклад, якщо $n = 13$, його двійкове представлення 1101_2. Це означає, що біноміальна купа з 13 вузлами складається з біноміальних дерев B_0, B_2 та B_3.

$(1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = 1 + 0 + 4 + 8 = 13)$.

Представлення біноміальної купи:

Біноміальна купа зазвичай представляється як зв'язаний список коренів її біноміальних дерев. Цей список може бути впорядкований за зростанням порядку дерев. Кожен вузол у біноміальному дереві містить:

- ключ (значення вузла)
- порядок (ступінь кореня, тобто порядок біноміального дерева, якщо це корінь)

– батько (покажчик на батьківський вузол)

– лівий_дочірній (покажчик на найлівіший дочірній вузол)

– правий_брат (покажчик на наступного брата, тобто на наступний дочірній вузол того ж батька)

1.3. Операції з Біноміальною купою

Розглянемо детально кожну з основних операцій, які можна виконувати над біноміальною купою .

MAKE-HEAP(): **Створення порожньої купи**

Ця операція є найпростішою. Вона просто ініціалізує нову, порожню біноміальну купу. Порожня купа не містить жодних біноміальних дерев.

Псевдокод такий:

```
MAKE-HEAP()
H.head = NIL // Встановлюємо покажчик на голову списку коренів як NIL
return H
```

H.head – це покажчик на перший корінь у зв’язаному списку коренів біноміальних дерев, що складають купу H. Встановлення його в NIL (нульовий покажчик) означає, що купа порожня і не містить жодних елементів.

MINIMUM(H): Пошук мінімального елемента

Оскільки біноміальна купа складається з кількох біноміальних дерев, і кожне з них задовольняє властивості мінімальної купи (найменший елемент знаходиться в корені), мінімальний елемент у всій купі буде одним з коренів біноміальних дерев. Ця операція сканує корені всіх біноміальних дерев у купі та повертає той, що має найменший ключ.

Псевдокод такий:

```
MINIMUM(H)
if H.head == NIL
return “Купа порожня”
min_node = H.head
current_node = H.head.right_sibling
while current_node != NIL
if current_node.key < min_node.key
min_node = current_node
```

```
current_node = current_node.right_sibling
return min_node
```
Ми починаємо з припущення, що перший корінь (H.head) є мінімальним. Потім ми проходимо по зв'язаному списку коренів (current_node = current_node.right_sibling), порівнюючи ключ кожного кореня з поточним мінімальним. Якщо знайдено менший ключ, оновлюємо min_node. Зрештою, min_node буде вказувати на вузол з найменшим ключем у всій купі.

Уявіть біноміальну купу як список коренів: $B_0(10)$, $B_2(5)$, $B_3(2)$.

3. min_node встановлюється на корінь $B_0(10)$.
4. Переходимо до $B_2(5)$. $5 < 10$, тому min_node оновлюється на $B_2(5)$.
5. Переходимо до $B_3(2)$. $2 < 5$, тому min_node оновлюється на $B_3(2)$.
6. Список закінчився. Повертаємо корінь $B_3(2)$, який містить мінімальне значення 2.

UNION(H1, H2): Об'єднання двох біноміальних куп

Операція об'єднання є однією з найважливіших і найскладніших операцій у біноміальних купах, оскільки багато інших операцій (таких як вставка та вилучення мінімуму) використовують її як допоміжну. Мета UNION(H1, H2) – об'єднати дві біноміальні купи H1 та H2 в одну нову біноміальну купу H, яка містить усі елементи з H1 та H2.

Процес об'єднання схожий на додавання двійкових чисел. Ми об'єднуємо списки коренів H1 та H2 у єдиний відсортований за порядком список. Потім ми проходимо по цьому об'єднаному списку, поєднуючи біноміальні дерева однакового порядку, дотримуючись правила, що в біноміальній купі може бути лише одне дерево кожного порядку.

Розглянемо допоміжну функцію: BINOMIAL-LINK(y, z)

Ця функція об'єднує два біноміальні дерева однакового порядку $k - 1$ в одне біноміальне дерево порядку k. Вона припускає, що y.key <= z.key. Якщо це не так, вузли y та z потрібно поміняти місцями. Вузол з більшим ключем стає дочірнім елементом вузла з меншим ключем.

Псевдокод BINOMIAL-LINK(y, z):

```
BINOMIAL-LINK(y, z)
// Припускаємо, що y.key <= z.key
```

```
// Якщо z.key < y.key, поміняйте y і z місцями
if z.key < y.key
swap(y, z) // Обміняти покажчики на вузли y та z
z.parent = y // z стає дочірнім елементом y
z.right_sibling = y.left_child // z стає найлівішим дочірнім елементо
м y
y.left_child = z // y.left_child тепер вказує на z
y.degree = y.degree + 1 // Збільшуємо ступінь y
return y // Повертаємо корінь нового дерева
Псевдокод UNION(H1, H2):
UNION(H1, H2)
H = MAKE-HEAP() // Створюємо нову порожню купу для результату
// 1. Злиття списків коренів H1 та H2
// Цей крок створює відсортований за порядком список коренів
// (злиття двох відсортованих списків)
H.head = MERGE-ROOT-LISTS(H1.head, H2.head)
if H.head == NIL
return H // Обидві купи були порожніми
prev_x = NIL
x = H.head
next_x = x.right_sibling
// 2. Прохід по об’єднаному списку та поєднання дерев однакового поря
дку
while next_x != NIL
// Випадок 1: Порядки x та next_x відрізняються
// АБО є три дерева з однаковим порядком (x, next_x, next_x.right_sib
ling)
// У цьому випадку ми просто переходимо до наступної пари
if (x.degree != next_x.degree) or   (next_x.right_sibling != NIL and
next_x.right_sibling.degree == x.degree)
prev_x = x
x = next_x
// Випадок 2: Порядки x та next_x однакові, і немає третього дерева з
цим порядком
// Об'єднуємо x та next_x
else if x.key <= next_x.key // Якщо ключ x менший або рівний ключу ne
xt_x
  x.right_sibling = next_x.right_sibling // next_x видаляється зі спи
ску
  BINOMIAL-LINK(x, next_x) // next_x стає дочірнім елементом x

// Випадок 3: Порядки x та next_x однакові, і ключ next_x менший за к
люч x
// Об'єднуємо next_x та x
else // next_x.key < x.key
```

```
  if prev_x != NIL
    prev_x.right_sibling = next_x // next_x стає наступним після prev
_x
  else
    H.head = next_x // next_x стає новим початком списку коренів

  x.right_sibling = next_x.right_sibling // x видаляється зі списку
  BINOMIAL-LINK(next_x, x) // x стає дочірнім елементом next_x
  x = next_x // Переходимо до нового кореня (next_x)

next_x = x.right_sibling // Переходимо до наступного елемента у списк
y
```
return H

Допоміжна функція: MERGE-ROOT-LISTS(h1_head, h2_head)

Ця функція зливає два відсортовані зв’язані списки коренів (за порядком дерева) в один відсортований список. Це стандартна операція злиття для зв’язаних списків.

Псевдокод MERGE-ROOT-LISTS(h1_head, h2_head):

```
MERGE-ROOT-LISTS(h1_head, h2_head)
result_head = NIL
tail = NIL
// Ініціалізація першого елемента
if h1_head == NIL
return h2_head
if h2_head == NIL
return h1_head
if h1_head.degree <= h2_head.degree
result_head = h1_head
h1_head = h1_head.right_sibling
else
result_head = h2_head
h2_head = h2_head.right_sibling
tail = result_head
// Злиття решти списків
while h1_head != NIL and h2_head != NIL
if h1_head.degree <= h2_head.degree
tail.right_sibling = h1_head
h1_head = h1_head.right_sibling
else
tail.right_sibling = h2_head
h2_head = h2_head.right_sibling
tail = tail.right_sibling
```

```
// Додавання залишків
if h1_head != NIL
tail.right_sibling = h1_head
else
tail.right_sibling = h2_head
return result_head
```
Розглянемо детальніше роботу UNION:

Припустимо, ми маємо дві біноміальні купи:

$H1$: $B_0(5)$, $B_2(12)$

$H2$: $B_0(3)$, $B_1(8)$, $B_2(7)$

1. Злиття списків коренів (MERGE-ROOT-LISTS):
    - Початкові списки:

– **H1**: `(5, deg 0)` → `(12, deg 2)`

- Тут `(5, deg 0)` – це корінь біноміального дерева порядку 0 з ключем 5.
- `(12, deg 2)` – це корінь біноміального дерева порядку 2 з ключем 12.

– **H2**: `(3, deg 0)` → `(8, deg 1)` → `(7, deg 2)`

- `(3, deg 0)` – корінь B_0 з ключем 3.
- `(8, deg 1)` – корінь B_1 з ключем 8.
- `(7, deg 2)` – корінь B_2 з ключем 7.

Процес злиття (`MERGE-ROOT-LISTS`):

Покроковий розбір `MERGE-ROOT-LISTS`

Вхідні дані:

– **`h1_head`** вказує на початок списку коренів H1: `(5, deg 0) -> (12, deg 2)`

– **`h2_head`** вказує на початок списку коренів H2: `(3, deg 0) -> (8, deg 1) -> (7, deg 2)`

Мета: Злити ці два списки коренів в один відсортований за порядком (ступенем) список, який стане новим **`H.head`** для об’єднаної купи.

Початкові значення змінних:

– `result_head = NIL` (покажчик на голову результуючого списку)

– `tail = NIL` (покажчик на останній елемент результуючого списку)

Крок 1: Ініціалізація першого елемента

2. `if h1_head == NIL`: Перевіряємо, чи `h1_head` порожній. Ні, він вказує на `(5, deg 0)`.
3. `if h2_head == NIL`: Перевіряємо, чи `h2_head` порожній. Ні, він вказує на `(3, deg 0)`.
4. `if h1_head.degree <= h2_head.degree`: Порівнюємо ступені перших елементів:
    - `h1_head.degree` це `0` (для вузла з ключем 5).
    - `h2_head.degree` це `0` (для вузла з ключем 3).
    - `0 <= 0` є істиною.
5. Виконується блок `if`:
    - `result_head = h1_head`: `result_head` тепер вказує на `(5, deg 0)`. Цей вузол стає першим у новому списку.
    - `h1_head = h1_head.right_sibling`: `h1_head` переміщується до наступного елемента в списку H1. Тепер `h1_head` вказує на `(12, deg 2)`.
    - Стан: `result_head: (5, deg 0)`, `h1_head: (12, deg 2)`, `h2_head: (3, deg 0)`
6. `tail = result_head`: `tail` також починає вказувати на `(5, deg 0)`.

Візуалізація стану після Кроку 1:

– `result_head` -> `(5, deg 0)`

– `tail` -> `(5, deg 0)`

– `h1_head` -> `(12, deg 2)`

– `h2_head` -> `(3, deg 0)`

Крок 2: Злиття решти списків (Цикл `while`)

Цикл виконуватиметься, доки обидва `h1_head` та `h2_head` не стануть `NIL`.

Ітерація 1:

7. `while h1_head != NIL and h2_head != NIL`:
    - `h1_head` є `(12, deg 2)` (не `NIL`).
    - `h2_head` є `(3, deg 0)` (не `NIL`).
    - Умова істинна.
8. `if h1_head.degree <= h2_head.degree`: Порівнюємо ступені поточних елементів:
    - `h1_head.degree` це `2` (для вузла з ключем 12).
    - `h2_head.degree` це `0` (для вузла з ключем 3).
    - `2 <= 0` є хибністю.
9. Виконується блок `else`:
    - `tail.right_sibling = h2_head`: Вузол, на який вказує `tail` (`(5, deg 0)`), тепер вказує своїм `right_sibling` на `h2_head` (`(3, deg 0)`). Це приєднує `(3, deg 0)` до результуючого списку.
        - Список тепер виглядає: `(5, deg 0) -> (3, deg 0)`
    - `h2_head = h2_head.right_sibling`: `h2_head` переміщується до наступного елемента в списку H2. Тепер `h2_head` вказує на `(8, deg 1)`.
    - `tail = tail.right_sibling`: `tail` переміщується, щоб вказувати на новододаний елемент. Тепер `tail` вказує на `(3, deg 0)`.

Візуалізація стану після Ітерації 1:

- `result_head` -> `(5, deg 0) -> (3, deg 0)`
- `tail` -> `(3, deg 0)`
- `h1_head` -> `(12, deg 2)`
- `h2_head` -> `(8, deg 1)`

Ітерація 2:

10. `while h1_head != NIL and h2_head != NIL`:
    - `h1_head` є `(12, deg 2)` (не `NIL`).
    - `h2_head` є `(8, deg 1)` (не `NIL`).
    - Умова істинна.
11. `if h1_head.degree <= h2_head.degree`: Порівнюємо ступені поточних елементів:
    - `h1_head.degree` це `2` (для вузла з ключем 12).
    - `h2_head.degree` це `1` (для вузла з ключем 8).
    - `2 <= 1` є хибністю.
12. Виконується блок `else`:
    - `tail.right_sibling = h2_head`: Вузол, на який вказує `tail` (`(3, deg 0)`), тепер вказує своїм `right_sibling` на `h2_head` (`(8, deg 1)`).
        - Список тепер виглядає: `(5, deg 0) -> (3, deg 0) -> (8, deg 1)`
    - `h2_head = h2_head.right_sibling`: `h2_head` переміщується. Тепер `h2_head` вказує на `(7, deg 2)`.
    - `tail = tail.right_sibling`: `tail` переміщується. Тепер `tail` вказує на `(8, deg 1)`.

Візуалізація стану після Ітерації 2:

– `result_head` -> `(5, deg 0) -> (3, deg 0) -> (8, deg 1)`

– `tail` > `(8, deg 1)`

– `h1_head` -> `(12, deg 2)`

– `h2_head` -> `(7, deg 2)`

Ітерація 3:

13. `while h1_head != NIL and h2_head != NIL`:

- `h1_head` є `(12, deg 2)` (не `NIL`).
- `h2_head` є `(7, deg 2)` (не `NIL`).
- Умова істинна.

14. `if h1_head.degree <= h2_head.degree`: Порівнюємо ступені поточних елементів:
    - `h1_head.degree` це `2` (для вузла з ключем 12).
    - `h2_head.degree` це `2` (для вузла з ключем 7).
    - `2 <= 2` є істиною.
15. Виконується блок `if`:
    - `tail.right_sibling = h1_head`: Вузол, на який вказує `tail` (`(8, deg 1)`), тепер вказує своїм `right_sibling` на `h1_head` (`(12, deg 2)`).
        - Список тепер виглядає: `(5, deg 0) -> (3, deg 0) -> (8, deg 1) -> (12, deg 2)`
    - `h1_head = h1_head.right_sibling`: `h1_head` переміщується. Тепер `h1_head` вказує на `NIL` (кінець списку H1).
    - `tail = tail.right_sibling`: `tail` переміщується. Тепер `tail` вказує на `(12, deg 2)`.

Візуалізація стану після Ітерації 3:

– `result_head` -> `(5, deg 0) -> (3, deg 0) -> (8, deg 1) -> (12, deg 2)`

– `tail` –> `(12, deg 2)`

– `h1_head` -> `NIL`

– `h2_head` -> `(7, deg 2)`

Крок 3: Додавання залишків

16. Цикл `while` завершився, оскільки `h1_head` є `NIL`.
17. `if h1_head != NIL`: Ця умова є хибністю.

18. Виконується блок `else`:

- `tail.right_sibling = h2_head`: Вузол, на який вказує `tail` (`(12, deg 2)`), тепер вказує своїм `right_sibling` на `h2_head` (`(7, deg 2)`). Це приєднує решту списку H2.

Фінальний стан результуючого списку:

`result_head` тепер вказує на список:

`(5, deg 0) -> (3, deg 0) -> (8, deg 1) -> (12, deg 2) -> (7, deg 2)`

Результат `MERGE-ROOT-LISTS`: Функція повертає покажчик на голову цього нового, об'єднаного та відсортованого за ступенем списку. Цей список потім буде використаний в операції `UNION` для подальшого поєднання біноміальних дерев однакового порядку.

Зверніть увагу, що в цьому списку є два дерева порядку 0 (`(5, deg 0)` та `(3, deg 0)`) і два дерева порядку 2 (`(12, deg 2)` та `(7, deg 2)`). Це нормально для результату `MERGE-ROOT-LISTS`. Наступний етап в операції `UNION` (`while next_x != NIL ...`) буде відповідальним за поєднання цих дерев однакового порядку, щоб забезпечити, що в фінальній біноміальній купі буде лише одне дерево кожного порядку.

Цей відсортований список є першим етапом у складній операції `UNION`, яка потім перейде до кроку поєднання біноміальних дерев з однаковими порядками.

Наведемо покроковий розбір `UNION(H1, H2)`

Початковий стан:

`H = MAKE-HEAP()`: Створено нову порожню купу `H`.

`H.head = MERGE-ROOT-LISTS(H1.head, H2.head)`:

З попереднього кроку, `H.head` тепер вказує на наступний список коренів:

`(5, deg 0) -> (3, deg 0) -> (8, deg 1) -> (12, deg 2) -> (7, deg 2)`

`if H.head == NIL`: Ні, `H.head` не `NIL`.

`prev_x = NIL`

`x = H.head` (тобто `(5, deg 0)`)

`next_x = x.right_sibling` (тобто `(3, deg 0)`)

Мета: Пройти по цьому списку, поєднуючи дерева однакового порядку, щоб у кінцевій купі `H` було щонайбільше одне дерево кожного порядку.

Крок 1: Ітерація циклу `while next_x != NIL`

Поточний стан:

`prev_x = NIL`

`x = (5, deg 0)`

`next_x = (3, deg 0)`

`H.head`:

`(5, deg 0) -> (3, deg 0) -> (8, deg 1) -> (12, deg 2) -> (7, deg 2)`

Перевірка умови `while`: `next_x` (`(3, deg 0)`) не `NIL`. Умова істинна.

Перевірка умов `if` / `else if` / `else`:

1. (`x.degree != next_x.degree`): `0 != 0` є хибністю.

2. `or (next_x.right_sibling != NIL and next_x.right_sibling.degree == x.degree)`:

- `next_x.right_sibling` є `(8, deg 1)`. Воно не `NIL`.
- `next_x.right_sibling.degree` є `1`.
- `x.degree` є `0`.
- `1 == 0` є хибністю.
- Отже, вся умова `if` є хибністю.

Переходимо до `else if`:

- `else if x.key <= next_x.key`: `x.key` (5) та `next_x.key` (3). `5 <= 3` є хибністю.

Переходимо до `else`: (Випадок 3: `next_x.key < x.key` і порядки однакові)

– `if prev_x != NIL`: `prev_x` є `NIL`. Умова хибна.

– Виконується блок `else`:

- `H.head = next_x`: `H.head` тепер вказує на `(3, deg 0)`. Це означає, що корінь `(3, deg 0)` стане новим початком списку коренів.

        - Список коренів `H.head` тепер виглядає так (логічно, фізично зв'язки змінюються далі):

          `(3, deg 0) -> (5, deg 0) -> (8, deg 1) -> (12, deg 2) -> (7, deg 2)`

- `x.right_sibling = next_x.right_sibling`: `(5, deg 0).right_sibling` тепер вказує на `(8, deg 1)`. Це "видаляє" `(3, deg 0)` зі списку після `(5, deg 0)`.
    - Список коренів `H.head` (після зміни `H.head` та `x.right_sibling`):

      `(3, deg 0)` (який був `next_x`)

      `(5, deg 0)` (який був `x`) тепер вказує на `(8, deg 1)`.
- `BINOMIAL-LINK(next_x, x)`: Викликаємо `BINOMIAL-LINK` з `y = (3, deg 0)` та `z = (5, deg 0)`.
    - Оскільки `next_x.key` (3) менший за `x.key` (5), `(5, deg 0)` стає дочірнім елементом `(3, deg 0)`.
    - `z.parent = y`: `(5, deg 0).parent` стає `(3, deg 0)`.
    - `z.right_sibling = y.left_child`: `(5, deg 0).right_sibling` стає `NIL` (якщо `(3, deg 0)` не мав дочірніх елементів).
    - `y.left_child = z`: `(3, deg 0).left_child` тепер вказує на `(5, deg 0)`.
    - `y.degree = y.degree + 1`: `(3, deg 0).degree` стає `1`.
    - Результат: `(3, deg 0)` перетворюється на `(3, deg 1)` з дочірнім `(5, deg 0)`.
- `x = next_x`: `x` тепер вказує на `(3, deg 1)` (новий корінь).
- `next_x = x.right_sibling`: `next_x` тепер вказує на `(8, deg 1)`.

Візуалізація стану після Ітерації 1:

- `prev_x = NIL`
- `x = (3, deg 1)` (це дерево B_1 з коренем 3, що має дочірній вузол 5)
- `next_x = (8, deg 1)`

- `H.head`:
  `(3, deg 1) -> (8, deg 1) -> (12, deg 2) -> (7, deg 2)`

(Зверніть увагу: `(5, deg 0)` тепер є дочірнім вузлом `(3, deg 1)` і більше не є коренем у списку `H.head`)

Крок 2: Ітерація циклу `while next_x != NIL`

Поточний стан:

- `prev_x = NIL`
- `x = (3, deg 1)`
- `next_x = (8, deg 1)`
- `H.head`: `(3, deg 1) -> (8, deg 1) -> (12, deg 2) -> (7, deg 2)`

Перевірка умови `while`: `next_x` (`(8, deg 1)`) не `NIL`. Умова істинна.

Перевірка умов `if` / `else if` / `else`:

1. `(x.degree != next_x.degree)`: `1 != 1` є хибністю.

2. `or (next_x.right_sibling != NIL and next_x.right_sibling.degree == x.degree)`:

- `next_x.right_sibling` є `(12, deg 2)`. Воно не `NIL`.
- `next_x.right_sibling.degree` є `2`.
- `x.degree` є `1`. * `2 == 1` є хибністю.
- Отже, вся умова `if` є хибністю.

Переходимо до `else if`:

`else if x.key <= next_x.key`: `x.key` (3) та `next_x.key` (8). `3 <= 8` є істиною.

Виконується блок `else if`:

(Випадок 2: `x.key <= next_x.key` і порядки однакові)

– `x.right_sibling = next_x.right_sibling`: `(3, deg 1).right_sibling` тепер вказує на `(12, deg 2)`. Це "видаляє" `(8, deg 1)` зі списку після `(3, deg 1)`.

  - Список коренів `H.head` тепер виглядає:

    `(3, deg 1) -> (12, deg 2) -> (7, deg 2)`

- **BINOMIAL-LINK(x, next_x)**: Викликаємо **BINOMIAL-LINK** з **y = (3, deg 1)** та **z = (8, deg 1)**.
  - Оскільки **y.key** (3) менший за **z.key** (8), **(8, deg 1)** стає дочірнім елементом **(3, deg 1)**.
  - **z.parent = y**: **(8, deg 1).parent** стає **(3, deg 1)**.
  - **z.right_sibling = y.left_child**: **(8, deg 1).right_sibling** стає **(5, deg 0)** (попередній **left_child** вузла 3).
  - **y.left_child = z**: **(3, deg 1).left_child** тепер вказує на **(8, deg 1)**.
  - **y.degree = y.degree + 1**: **(3, deg 1).degree** стає **2**.
  - Результат: **(3, deg 1)** перетворюється на **(3, deg 2)** з дочірніми **(8, deg 1)** та **(5, deg 0)**.
- **next_x = x.right_sibling**: **next_x** тепер вказує на **(12, deg 2)**.

Візуалізація стану після Ітерації 2:

- **prev_x = NIL**
- **x = (3, deg 2)** (це дерево B_2 з коренем 3, що має дочірні вузли 8 та 5)
- **next_x = (12, deg 2)**
- **H.head**: **(3, deg 2) -> (12, deg 2) -> (7, deg 2)**

Крок 3: Ітерація циклу while next_x != NIL

Поточний стан:

prev_x = NIL

x = (3, deg 2)

next_x = (12, deg 2)

H.head: **(3, deg 2) -> (12, deg 2) -> (7, deg 2)**

Перевірка умови **while**: **next_x** (**(12, deg 2)**) не **NIL**. Умова істинна.

Перевірка умов **if** / **else if** / **else**:

1. **(x.degree != next_x.degree)**: **2 != 2** є хибністю.

2. `or (next_x.right_sibling != NIL and next_x.right_sibling.degree == x.degree)`:

- `next_x.right_sibling` є `(7, deg 2)`. Воно не `NIL`.
- `next_x.right_sibling.degree` є `2`.
- `x.degree` є `2`. * `2 == 2` є істиною.
- Отже, вся умова `if` є істиною. (Випадок 1: Три дерева з однаковим порядком).

Виконується блок `if`:

– `prev_x = x`: `prev_x` тепер вказує на `(3, deg 2)`.

– `x = next_x`: `x` тепер вказує на `(12, deg 2)`.

– `next_x = x.right_sibling`: `next_x` тепер вказує на `(7, deg 2)`.

Візуалізація стану після Ітерації 3:

- `prev_x = (3, deg 2)`
- `x = (12, deg 2)`
- `next_x = (7, deg 2)`
- `H.head`: `(3, deg 2) -> (12, deg 2) -> (7, deg 2)` (список коренів не змінився на цьому кроці)

Крок 4: Ітерація циклу `while next_x != NIL`

Поточний стан:

- `prev_x = (3, deg 2)`
- `x = (12, deg 2)`
- `next_x = (7, deg 2)`
- `H.head`: `(3, deg 2) -> (12, deg 2) -> (7, deg 2)`

Перевірка умови `while`: `next_x` (`(7, deg 2)`) не `NIL`. Умова істинна.

Перевірка умов `if` / `else if` / `else`:

1. `(x.degree != next_x.degree)`: `2 != 2` є хибністю.

2. `or (next_x.right_sibling != NIL and next_x.right_sibling.degree == x.degree)`:

- `next_x.right_sibling` є `NIL`.
- Отже, вся умова `if` є хибністю.

Переходимо до `else if`:

`else if x.key <= next_x.key`: `x.key` (12) та `next_x.key` (7). `12 <= 7` є хибністю.

Переходимо до `else`: (Випадок 3: `next_x.key < x.key` і порядки однакові)

- `if prev_x != NIL`: `prev_x` є `(3, deg 2)`. Умова істинна.
  - `prev_x.right_sibling = next_x`: `(3, deg 2).right_sibling` тепер вказує на `(7, deg 2)`. Це "видаляє" `(12, deg 2)` зі списку після `(3, deg 2)`.
    - Список коренів `H.head` тепер виглядає:

      `(3, deg 2) -> (7, deg 2)` (потім буде приєднано результат `BINOMIAL-LINK`)
- `x.right_sibling = next_x.right_sibling`: `(12, deg 2).right_sibling` тепер вказує на `NIL`.
- `BINOMIAL-LINK(next_x, x)`: Викликаємо `BINOMIAL-LINK` з `y = (7, deg 2)` та `z = (12, deg 2)`.
  - Оскільки `next_x.key` (7) менший за `x.key` (12), `(12, deg 2)` стає дочірнім елементом `(7, deg 2)`.
  - `z.parent = y`: `(12, deg 2).parent` стає `(7, deg 2)`.
  - `z.right_sibling = y.left_child`: `(12, deg 2).right_sibling` стає `NIL` (якщо `(7, deg 2)` не мав дочірніх елементів).
  - `y.left_child = z`: `(7, deg 2).left_child` тепер вказує на `(12, deg 2)`.
  - `y.degree = y.degree + 1`: `(7, deg 2).degree` стає `3`.
  - Результат: `(7, deg 2)` перетворюється на `(7, deg 3)` з дочірнім `(12, deg 2)`.
- `x = next_x`: `x` тепер вказує на `(7, deg 3)` (новий корінь).
- `next_x = x.right_sibling`: `next_x` тепер вказує на `NIL` (оскільки `(7, deg 3)` не має правого брата).

Візуалізація стану після Ітерації 4:

- `prev_x = (3, deg 2)`
- `x = (7, deg 3)` (це дерево B_3 з коренем 7, що має дочірній вузол 12)
- `next_x = NIL`
- `H.head`: `(3, deg 2) -> (7, deg 3)`

Крок 5: Завершення циклу

Перевірка умови `while`: `next_x` є `NIL`. Умова хибна. Цикл завершується.

Кінцевий результат `UNION(H1, H2)`:

Функція повертає купу `H`, `H.head` якої вказує на список коренів:

`(3, deg 2)` (це біноміальне дерево B_2 з коренем 3, що має дочірні вузли 8 та 5)

`->`

`(7, deg 3)` (це біноміальне дерево B_3 з коренем 7, що має дочірній вузол 12)

Фінальна біноміальна купа `H` складається з двох біноміальних дерев: B_2 та B_3.

- Дерево B_2 має корінь 3. Його дочірні елементи: 8 (корінь B_1) та 5 (корінь B_0).
- Дерево B_3 має корінь 7. Його дочірні елементи: 12 (корінь B_2).

Ця купа містить всі початкові елементи: {5,12} з H1 та {3,8,7} з H2, а також 10, якщо це був дочірній елемент 5.

Загальна кількість вузлів: $2^2 + 2^3 = 4 + 8 = 12$.

(Якщо 10 був дочірнім 5, то 5 мав бути коренем B_1, а не B_0. У прикладі було `(5, deg 0)`, тому 5 є B_0. Припустимо, що 10 не було в початкових даних, або це була помилка в описі. Якщо ж 5 був коренем B_1, то він мав би 2 вузли. Давайте дотримуватись `(5, deg 0)` як B_0 з 1 вузлом).

Початкові елементи:

H1: 5 (B0), 12 (B2) -> 1 + 4 = 5 вузлів

H2: 3 (B0), 8 (B1), 7 (B2) -> 1 + 2 + 4 = 7 вузлів

Всього: 5 + 7 = 12 вузлів.

Кінцева купа:

B2 (корінь 3) -> 4 вузли

B3 (корінь 7) -> 8 вузлів

Всього: 4 + 8 = 12 вузлів.

Структура купи відповідає двійковому представленню числа 12: 1100_2, що означає B_2 та B_3.

INSERT(H, x): **Вставка нового елемента**

Щоб вставити новий елемент x у біноміальну купу H, ми створюємо нову біноміальну купу, що містить лише один вузол x (це біноміальне дерево B_0 з ключем x.key). Потім ми об'єднуємо цю нову купу з оригінальною купою H за допомогою операції UNION.

Псевдокод:

```
INSERT(H, x)
H_prime = MAKE-HEAP() // Створюємо нову купу для x
x.parent = NIL
x.left_child = NIL
x.right_sibling = NIL
x.degree = 0
H_prime.head = x // x стає єдиним коренем у H_prime (B_0)
H = UNION(H, H_prime) // Об'єднуємо H з H_prime
return H
```

Ця операція демонструє потужність UNION. Вставка елемента зводиться до об'єднання купи з новою купою , що містить лише один елемент. Амортизована складність вставки є $O(\log n)$, оскільки $UNION$ має таку складність.

Вставимо елемент 6 у купу H: $B_2(3)$, $B_3(7)$.

1. Створюємо H_new з одним вузлом (6, deg 0).
2. Викликаємо UNION(H, H_new).
    - H: $B_2(3)$, $B_3(7)$
    - H_new: $B_0(6)$
    - Злиття списків коренів: (6, deg 0)to(3, deg 2)to(7, deg 3)

- Прохід:
    - x=(6, deg 0), next_x=(3, deg 2). Порядки різні. Переходимо далі.
    - prev_x=(6, deg 0), x=(3, deg 2), next_x=(7, deg 3). Порядки різні. Переходимо далі.
    - prev_x=(3, deg 2), x=(7, deg 3), next_x=NIL. Кінець.
- Результат: H тепер містить $B_0(6)$, $B_2(3)$, $B_3(7)$.

EXTRACT-MIN(H): **Вилучення мінімального елемента**

Ця операція видаляє елемент з найменшим ключем з біноміальної купи та повертає його. Це одна з найважливіших операцій для купи пріоритетів.

Кроки:

1. Знайти мінімальний корінь: Спочатку знаходимо корінь біноміального дерева, який містить мінімальний ключ у всій купі. Це робиться за допомогою сканування списку коренів, як у операції MINIMUM.
2. Видалити мінімальний корінь: Видаляємо знайдений мінімальний корінь зі списку коренів біноміальної купи H.
3. Створити нову купу з дочірніх елементів: Дочірні елементи видаленого кореня самі є коренями біноміальних дерев. Ці дочірні елементи, якщо їх розглядати окремо, утворюють нову біноміальну купу. Важливо, що вони розташовані у порядку зменшення ступеня.
4. Об’єднати купи: Об’єднуємо оригінальну купу H (після видалення мінімального кореня) з цією новоствореною купою дочірніх елементів за допомогою операції UNION.

Псевдокод такий:

```
EXTRACT-MIN(H)
if H.head == NIL
return “Купа порожня”
// 1. Знайти мінімальний корінь
min_node_prev = NIL
min_node = H.head
current_node = H.head.right_sibling
current_node_prev = H.head
```

```
while current_node != NIL
if current_node.key < min_node.key
min_node = current_node
min_node_prev = current_node_prev
current_node_prev = current_node
current_node = current_node.right_sibling
// 2. Видалити мінімальний корінь зі списку коренів H
if min_node_prev == NIL // min_node був першим у списку
H.head = min_node.right_sibling
else
min_node_prev.right_sibling = min_node.right_sibling
// 3. Створити нову купу H_prime з дочірніх елементів min_node
H_prime = MAKE-HEAP()
// Дочірні елементи min_node розташовані у порядку зменшення ступеня.
// Нам потрібно обернути їх порядок, щоб вони були за зростанням ступ
еня
// для коректного злиття в MERGE-ROOT-LISTS.
child = min_node.left_child
while child != NIL
next_child = child.right_sibling
child.right_sibling = H_prime.head // Додаємо дочірній елемент на поч
аток H_prime
child.parent = NIL // Очищаємо батьківський покажчик
H_prime.head = child
child = next_child
// 4. Об’єднати H з H_prime
H = UNION(H, H_prime)
return min_node // Повертаємо вилучений вузол
```

Крок 3, де ми створюємо H_prime, вимагає особливої уваги. Коли ми “від’єднуємо” корінь, його дочірні елементи, які були пов’язані як брати, утворюють новий список коренів. За визначенням біноміального дерева, ці дочірні елементи є коренями $B_k - 1, B_k - 2, \ldots, B_0$ і розташовані у порядку зменшення ступеня.

Для коректного злиття за допомогою MERGE-ROOT-LISTS (яка очікує списки, відсортовані за зростанням ступеня), ми повинні обернути цей список дочірніх елементів. Це робиться шляхом послідовного додавання кожного дочірнього елемента на початок списку H_prime.head.

Розглянемо детальніше EXTRACT-MIN:

Припустимо, у нас є купа H: $B_0(6)$, $B_2(3)$, $B_3(7)$.

(Мінімальний елемент 3 знаходиться в корені B_2)

1. Знаходимо мінімальний корінь: min_node вказує на корінь $B_2(3)$. min_node_prev вказує на корінь $B_0(6)$.
2. Видаляємо мінімальний корінь: $B_2(3)$ видаляється зі списку коренів.
    - Список коренів H тепер: $B_0(6) \rightarrow B_3(7)$.
3. Створюємо нову купу з дочірніх елементів $B_2(3)$:
    - Дочірні елементи $B_2(3)$ – це корені B_1 та B_0. Припустимо, це $B_1(5)$ та $B_0(10)$ (де 5 було дочірнім 3, а 10 було дочірнім 5).
    - Вони розташовані у порядку зменшення ступеня: $B_1(5) \rightarrow B_0(10)$.
    - Ми обертаємо цей список, щоб створити H_prime: $B_0(10) \rightarrow B_1(5)$.
    - H_prime тепер містить $B_0(10)$ та $B_1(5)$.
4. Об’єднуємо H та H_prime: UNION(H, H_prime)
    - H: $B_0(6)$, $B_3(7)$
    - H_prime: $B_0(10)$, $B_1(5)$
    - Злиття списків коренів: (6, deg 0)$(10, deg 0)to(5, deg 1)$(7, deg 3)
    - Прохід та поєднання:
        - x=(6, deg 0), next_x=(10, deg 0). Об’єднуємо. $6 \leq 10$. BINOMIAL-LINK((6, deg 0), (10, deg 0)).
            - 10 стає дочірнім 6. 6 стає $B_1(6)$.
            - Список: (6, deg 1)to(5, deg 1)to(7, deg 3)
        - x=(6, deg 1), next_x=(5, deg 1). Об’єднуємо. $5 < 6$. BINOMIAL-LINK((5, deg 1), (6, deg 1)).
            - 6 стає дочірнім 5. 5 стає $B_2(5)$.
            - Список: (5, deg 2)to(7, deg 3)
        - x=(5, deg 2), next_x=(7, deg 3). Порядки різні. Переходимо далі.
        - Кінець.

- Результат: H тепер містить $B_2(5)$, $B_3(7)$.

DECREASE-KEY(H, x, k): **Зменшення ключа елемента**

Ця операція зменшує значення ключа вузла x до нового значення k. Важливо, що k повинно бути меншим за поточний ключ x.key. Після зменшення ключа, властивість мінімальної купи може бути порушена, тому необхідно відновити її. Це робиться шляхом “бульбашкового” підняття вузла x вгору по дереву, обмінюючи його з батьківським вузлом, доки властивість купи не буде відновлена або доки вузол x не стане коренем.

Псевдокод:

```
DECREASE-KEY(H, x, k)
if k > x.key
return “Новий ключ більший за поточний”
x.key = k // Зменшуємо ключ
y = x
z = y.parent
// Піднімаємо вузол y вгору, доки властивість купи не буде відновлена
while z != NIL and y.key < z.key
// Обмінюємо ключі y та z
temp_key = y.key
y.key = z.key
z.key = temp_key
// Якщо вузли мають додаткові дані, їх також потрібно обміняти,
// або краще, обмінювати покажчики на вузли, а не лише ключі.
// Для простоти псевдокоду, ми обмінюємо ключі. У реалізації
// краще обмінювати покажчики на вузли, щоб зберегти структуру.

// Продовжуємо підніматися вгору
y = z
z = y.parent
```

Алгоритм працює за принципом “бульбашкового підняття”. Після зменшення ключа x, ми порівнюємо його з ключем його батьківського вузла z. Якщо ключ x менший, ніж ключ z, ми обмінюємо їхні ключі. Цей процес повторюється, доки x не стане коренем дерева або доки його ключ не стане більшим або рівним ключу його батька. Це гарантує, що властивість мінімальної купи відновлюється вздовж шляху від x до кореня.

Візуалізація DECREASE-KEY:

Припустимо, у нас є біноміальне дерево B_3 з коренем 7.

```
    (7)
   / |  \
```

```
  (8)(12)(15)
   / |   |
(10)(11)(13)
 |
(14)
```

Зменшимо ключ вузла 14 до 2. (DECREASE-KEY(H, вузол_14, 2))

5. Ключ вузла 14 стає 2.

```
   (7)
  / | \
(8)(12)(15)
 /  |   |
(10)(11)(13)
|
  (2) <- Змінено
```

6. Порівнюємо (2) з його батьком (10). $2 < 10$. Обмінюємо ключі.

```
   (7)
  / | \
(8)(12)(15)
  / | |
(2)(11)(13) <- Обміняно
 |
(10)
```

7. Порівнюємо (2) з його новим батьком (8). $2 < 8$. Обмінюємо ключі.

```
   (7)
  / |  \
(2)(12)(15) <- Обміняно
 /   |  |
(8)(11)(13)
 |
(10)
```

8. Порівнюємо (2) з його новим батьком (7). $2 < 7$. Обмінюємо ключі.

```
   (2) <- Обміняно, тепер корінь
  / |  \
(7)(12)(15)
 /  |   |
(8)(11)(13)
 |
(10)
```

9. Вузол з ключем 2 тепер є коренем дерева. Процес завершено.

DELETE(H, x): **Видалення елемента**

Видалення довільного елемента x з біноміальної купи виконується у два етапи:

1. Зменшити ключ до мінімуму: Зменшуємо ключ вузла x до найменшого можливого значення (наприклад, до –infty). Це гарантує, що x стане мінімальним елементом у своєму біноміальному дереві, і, можливо, у всій купі.
2. Вилучити мінімальний елемент: Після зменшення ключа, викликаємо операцію EXTRACT-MIN(H). Оскільки x тепер має найменший ключ, саме він буде вилучений.

Псевдокод:

```
DELETE(H, x)
// Припускаємо, що MIN_INF - це значення, менше за будь-який можливий ключ у купі.
// Наприклад, для цілих чисел це може бути дуже велике від'ємне число
DECREASE-KEY(H, x, MIN_INF) // Зменшуємо ключ x до мінімуму
EXTRACT-MIN(H) // Вилучаємо мінімальний елемент (яким тепер є x)
```

Ця операція є зручним способом видалення довільного елемента, використовуючи вже існуючі операції DECREASE-KEY та EXTRACT-MIN. Складність операції DELETE буде такою ж, як і складність EXTRACT-MIN, тобто O(logn), оскільки DECREASE-KEY має складність O(logn).

Візуалізація DELETE:

Видалимо елемент 11 з купи H: $B_2(5)$, $B_3(7)$ (де 11 знаходиться в B_3(7)).

```
     (7)
    / |  \
  (8)(12)(15)
   /  |   |
(10)(11)(13) <– Видалити 11
 |
(14)
```

1. DECREASE-KEY(H, вузол_11, MIN_INF):
    - Ключ вузла 11 стає –infty.
    - Вузол з –infty піднімається до кореня B_3.

```
  (-inf)
  / |  \
(7)(12)(15)
```

```
 /  |   |
(8)(11)(13) <– Колишній 11, тепер 7
 |
(10)
```

У візуалізації ключі переміщуються, але в реальності ми обмінюємо вузли або їх вміст, щоб зберегти посилання.

2. EXTRACT-MIN(H):

    - MINIMUM(H) знайде корінь з –infty як мінімальний.
    - Цей корінь видаляється зі списку коренів купи.
    - Його дочірні елементи (колишні корені B_2,B_1,B_0) утворюють нову купу.
    - Ця нова купа об’єднується з рештою H.
    - Вузол, який спочатку містив 11, а потім –infty, тепер вилучений.

Обходи дерева

Хоча біноміальна купа є лісом біноміальних дерев, обходи зазвичай застосовуються до окремого біноміального дерева. Ці обходи дозволяють відвідати кожен вузол дерева в певному порядку.

Представлення вузла для обходів:

Кожен вузол має покажчики на left_child (найлівіший дочірній) та right_sibling (правий брат). Це дозволяє нам переміщатися по дереву.

Обхід в глибину

Обходи в глибину досліджують гілку настільки глибоко, наскільки це можливо, перш ніж повертатися назад.

> **Прямий обхід.** Відвідує корінь, потім рекурсивно обходить його найлівіше дочірнє піддерево, а потім рекурсивно обходить піддерева його правих братів.

Псевдокод такий:

```
PRE-ORDER-TRAVERSAL(node)
if node != NIL
print node.key // Відвідати вузол (наприклад, вивести ключ)
// Обхід найлівішого дочірнього піддерева
PRE-ORDER-TRAVERSAL(node.left_child)
```

```
// Обхід піддерев правих братів
sibling = node.left_child.right_sibling
while sibling != NIL
  PRE-ORDER-TRAVERSAL(sibling)
  sibling = sibling.right_sibling
```

Цей псевдокод є дещо спрощеним для біноміальних дерев, оскільки left_child є коренем найстаршого дочірнього дерева, а його right_sibling – наступним дочірнім деревом. Більш загальний підхід для довільних дерев:

```
PRE-ORDER-TRAVERSAL_GENERIC(node)
if node != NIL
print node.key // Відвідати вузол
child = node.left_child // Починаємо з найлівішого дочірнього
while child != NIL
  PRE-ORDER-TRAVERSAL_GENERIC(child)
  child = child.right_sibling // Переходимо до наступного брата
```

Цей загальний псевдокод є більш коректним для структури біноміального дерева, де дочірні елементи пов’язані через right_sibling.

Зворотний обхід. Рекурсивно обходить найлівіше дочірнє піддерево, потім рекурсивно обходить піддерева його правих братів, і лише потім відвідує корінь.

Псевдокод:

POST-ORDER-TRAVERSAL(node)

if node != NIL

child = node.left_child

while child != NIL

POST-ORDER-TRAVERSAL(child)

child = child.right_sibling

```
print node.key // Відвідати вузол
```

Симетричний обхід. Цей обхід зазвичай застосовується до бінарних дерев і не має прямого аналога для біноміальних дерев через їхню багаторозгалужену структуру. Однак, якщо ми інтерпретуємо

“симетричний” як відвідування кореня між обходом його першого дочірнього піддерева та решти, це може виглядати так:

Псевдокод (адаптований для біноміальних дерев):

```
IN-ORDER-TRAVERSAL_ADAPTED(node)
if node != NIL
if node.left_child != NIL
IN-ORDER-TRAVERSAL_ADAPTED(node.left_child) // Обхід найлівішого дочірнього
print node.key // Відвідати корінь

// Обхід решти дочірніх елементів (братів найлівішого)
sibling = node.left_child.right_sibling
while sibling != NIL
  IN-ORDER-TRAVERSAL_ADAPTED(sibling)
  sibling = sibling.right_sibling
```

Цей “симетричний” обхід для біноміальних дерев є нетрадиційним і не має такого ж значення, як для бінарних дерев (де він виводить елементи у відсортованому порядку).

Обхід в ширину

Цей обхід відвідує всі вузли на одному рівні (глибині) перед тим, як перейти до вузлів на наступному рівні. Для цього зазвичай використовується черга.

Псевдокод:

```
LEVEL-ORDER-TRAVERSAL(root_node)
if root_node == NIL
return
Q = new Queue() // Створюємо нову чергу
Q.enqueue(root_node) // Додаємо корінь до черги
while not Q.is_empty()
current_node = Q.dequeue() // Вилучаємо вузол з черги
print current_node.key // Відвідати вузол
// Додаємо всіх дочірніх елементів поточного вузла до черги
child = current_node.left_child
while child != NIL
  Q.enqueue(child)
  child = child.right_sibling
```

Ми починаємо з додавання кореня дерева до черги. Потім, доки черга не порожня, ми вилучаємо вузол, відвідуємо його, а потім додаємо всіх його

дочірніх елементів до черги. Завдяки FIFO (First-In, First-Out) властивості черги, ми гарантуємо, що всі вузли на поточному рівні будуть відвідані перед тим, як ми перейдемо до вузлів на наступному рівні.

Область застосування

Біноміальні купи є ефективною структурою даних для реалізації пріоритетних черг, особливо коли операція об'єднання (злиття) є частою або критичною за продуктивністю. Їхня амортизована ефективність робить їх привабливим вибором для багатьох алгоритмів.

Основні області застосування включають:

1. Пріоритетні черги: Це найочевидніше і найпоширеніше застосування. Біноміальні купи можуть ефективно підтримувати операції вставки, вилучення мінімуму, зменшення ключа та об'єднання, що робить їх ідеальними для систем, де елементи обробляються на основі їхнього пріоритету.

2. Алгоритми на графах:

    - Алгоритм Дейкстри (Dijkstra's Algorithm): Використовується для знаходження найкоротших шляхів від одного джерела до всіх інших вершин у зваженому графі. Пріоритетна черга в алгоритмі Дейкстри використовується для ефективного вибору наступної вершини для обробки. Біноміальні купи можуть забезпечити ефективне оновлення відстаней (операція DECREASE-KEY) та вилучення мінімальної відстані (EXTRACT-MIN).

    - Алгоритм Прима (Prim's Algorithm): Використовується для знаходження мінімального остовного дерева у зваженому неорієнтованому графі. Пріоритетна черга тут допомагає вибрати наступне ребро з мінімальною вагою, яке з'єднує дерево з новою вершиною. Операції INSERT, EXTRACT-MIN та DECREASE-KEY є ключовими.

3. Імітаційне моделювання (Simulation): У дискретних подіях, де події відбуваються в певному порядку (наприклад, за часом), біноміальні купи можуть використовуватися для управління чергою подій, де подія з найменшим часом обробляється першою.

4. Мережеві маршрутизатори: Управління пакетами даних на основі їхнього пріоритету або часу прибуття.

5. Операційні системи: Планування завдань, де завдання з вищим пріоритетом виконуються першими.

Переваги та недоліки

Як і будь-яка структура даних, біноміальні купи мають свої сильні та слабкі сторони.

Переваги:

1. Ефективна операція об’єднання (UNION): Це одна з ключових переваг. Об’єднання двох біноміальних куп займає амортизований час $O(\logn)$, що є значно кращим, ніж у звичайних бінарних куп (O(n)). Ця ефективність робить біноміальні купи ідеальними для алгоритмів, які часто потребують злиття пріоритетних черг.

2. Амортизована ефективність: Більшість операцій (вставка, вилучення мінімуму, зменшення ключа) мають амортизовану часову складність $O(\logn)$. Це означає, що хоча окрема операція може зайняти більше часу (наприклад, при об’єднанні багатьох дерев), послідовність операцій буде виконуватися в середньому швидко.

3. Підтримка DECREASE-KEY: Ефективна операція зменшення ключа ($O(\logn)$ амортизовано) є критично важливою для багатьох алгоритмів на графах, таких як алгоритми Дейкстри та Прима.

4. Простота реалізації порівняно з Фібоначчієвими купами: Хоча біноміальні купи складніші за бінарні, вони значно простіші в реалізації порівняно з Фібоначчієвими купами, які пропонують кращу теоретичну амортизовану складність для деяких операцій, але є набагато складнішими.

Недоліки:

1. Складність реалізації порівняно з бінарними купами: Біноміальні купи вимагають більш складної реалізації через їхню структуру як лісу біноміальних дерев та складність операції UNION. Це може призвести до більшої кількості помилок при кодуванні.

2. Необхідність підтримки покажчиків на батьківські вузли: Для ефективної операції DECREASE-KEY та інших операцій, які

вимагають переміщення вгору по дереву, кожен вузол повинен мати покажчик на свого батька. Це збільшує обсяг пам’яті, необхідний для зберігання кожного вузла.

3. Не найшвидші в найгіршому випадку: Хоча амортизована складність біноміальних куп є хорошою, деякі операції в найгіршому випадку можуть бути повільнішими, ніж у інших структур (наприклад, EXTRACT-MIN може бути O(logn), але може включати багато BINOMIAL-LINK операцій). Для гарантованих часових обмежень, можливо, краще підійдуть інші структури.

4. Пошук довільного елемента: Операція SEARCH (пошук довільного елемента за його значенням) у біноміальній купі займає O(n) часу в найгіршому випадку, оскільки немає швидкого способу знайти елемент, крім як переглянути всі вузли. Це не є проблемою для пріоритетних черг, де доступ до елементів зазвичай здійснюється через покажчики, отримані під час вставки.

Контрольні запитання

1. Що таке біноміальне дерево порядку k і які його основні властивості (кількість вузлів, висота, ступінь кореня)?

2. Які дві ключові властивості визначають біноміальну купу? Поясніть, чому вони важливі.

3. Опишіть, як біноміальна купа представляється в пам’яті.

4. Чому операція UNION є центральною для біноміальних куп і як вона працює?

5. Поясніть, як операції INSERT, EXTRACT-MIN та DELETE використовують операцію UNION або її допоміжні функції.

6. Яка амортизована часова складність основних операцій біноміальної купи (INSERT, MINIMUM, EXTRACT-MIN, DECREASE-KEY, DELETE, UNION)?

7. Наведіть приклади алгоритмів, де ефективність біноміальних куп може бути перевагою.

8. Які недоліки біноміальних куп порівняно з іншими структурами даних?

9. Опишіть принцип роботи прямого обходу (pre-order traversal) для біноміального дерева.
10. Як обхід в ширину (level-order traversal) використовує чергу для відвідування вузлів?

Методичні вказівки до лабораторної роботи: Біноміальні купи

1. Назва

Реалізація та дослідження ефективності операцій біноміальної купи.

2. Мета

Ознайомлення зі структурою даних “Біноміальна купа”, здобуття практичних навичок її реалізації та дослідження часової складності основних операцій.

3. Завдання

1. Розробити програмну реалізацію біноміальної купи з підтримкою всіх основних операцій: MAKE-HEAP, MINIMUM, INSERT, EXTRACT-MIN, DECREASE-KEY, DELETE, UNION.
2. Реалізувати обходи біноміального дерева: прямий (pre-order), зворотний (post-order) та в ширину (level-order).
3. Провести експериментальне дослідження часової складності реалізованих операцій для різних обсягів даних.
4. Продемонструвати роботу біноміальної купи на прикладах.

4. Опис

Дана лабораторна робота передбачає глибоке занурення у світ біноміальних куп – потужної структури даних, що належить до сімейства пріоритетних черг. На відміну від простих бінарних куп, біноміальні купи є лісом біноміальних дерев, що забезпечує їм унікальні властивості та амортизовану ефективність для таких операцій, як об’єднання.

Студентам необхідно буде реалізувати структуру вузла біноміального дерева, що включає поля для ключа, ступеня, покажчиків на батька, найлівішого дочірнього елемента та правого брата. Сама купа буде представлена як зв’язаний список коренів біноміальних дерев. Особливу увагу слід приділити коректній реалізації операції UNION, оскільки вона є основою для багатьох інших операцій.

Експериментальне дослідження включатиме вимірювання часу виконання кожної операції для різних розмірів купи (наприклад, N=100,1000,10000,100000 елементів) та порівняння отриманих результатів з теоретичною амортизованою складністю O(logN).

5. Порядок виконання

1. Вивчення теоретичного матеріалу: Ретельно ознайомтесь з теоретичними відомостями про біноміальні дерева та біноміальні купи, їхні властивості та принципи роботи кожної операції. Зверніть увагу на структуру вузлів та представлення купи.
2. Вибір мови програмування: Оберіть мову програмування для реалізації (наприклад, C++, Java, Python).
3. Реалізація структури вузла та купи:
    - Визначте клас або структуру для представлення вузла біноміального дерева (з полями key, degree, parent, left_child, right_sibling).
    - Визначте клас або структуру для біноміальної купи (з полем head – покажчиком на голову списку коренів).
4. Реалізація допоміжної функції BINOMIAL-LINK(y, z): Ця функція є критично важливою для об’єднання біноміальних дерев. Пам’ятайте про властивість мінімальної купи.
5. Реалізація допоміжної функції MERGE-ROOT-LISTS(h1_head, h2_head): Злиття двох відсортованих списків коренів.
6. Реалізація операції UNION(H1, H2): Це найскладніша операція. Реалізуйте її згідно з псевдокодом, використовуючи MERGE-ROOT-LISTS та BINOMIAL-LINK.
7. Реалізація основних операцій купи:
    - MAKE-HEAP()
    - MINIMUM(H)
    - INSERT(H, x) (використовує UNION)
    - EXTRACT-MIN(H) (використовує UNION)

- DECREASE-KEY(H, x, k)
- DELETE(H, x) (використовує DECREASE-KEY та EXTRACT-MIN)

8. Реалізація обходів дерева:
    - PRE-ORDER-TRAVERSAL(node)
    - POST-ORDER-TRAVERSAL(node)
    - LEVEL-ORDER-TRAVERSAL(node) (використовує чергу)
9. Тестування та налагодження: Ретельно протестуйте кожну операцію на невеликих прикладах, щоб переконатися в її коректності. Використовуйте візуалізацію (на папері або за допомогою простого виведення структури) для перевірки.
10. Експериментальне дослідження:
    - Напишіть код для вимірювання часу виконання операцій.
    - Проведіть серію експериментів:
        - Вставка N випадкових елементів.
        - Вилучення N/2 мінімальних елементів.
        - Зменшення ключів для N/4 випадкових елементів.
        - Об’єднання двох куп з N/2 елементів кожна.
    - Зафіксуйте час виконання для різних значень N (наприклад, 102,103,104,105,106).
    - Побудуйте графіки залежності часу виконання від N для кожної операції.
    - Порівняйте отримані результати з теоретичними оцінками складності.
11. Оформлення звіту: Підготуйте звіт, що включає:
    - Опис реалізованої структури даних.
    - Псевдокоди реалізованих операцій.

- Результати експериментального дослідження (таблиці, графіки).
- Висновок щодо ефективності біноміальних куп.

6. Результат

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де реалізовано всі операції біноміальної купи (MAKE-HEAP, MINIMUM, INSERT, EXTRACT-MIN, DECREASE-KEY, DELETE, UNION) та обходи біноміального дерева (прямий, зворотний, в ширину). Код повинен бути добре прокоментований.
2. Продемонструвати роботу додатку:
    - Створення купи.
    - Послідовну вставку кількох елементів та демонстрацію зміни структури купи (наприклад, виведення коренів та їхніх ступенів).
    - Вилучення мінімального елемента.
    - Зменшення ключа елемента та демонстрацію відновлення властивості купи.
    - Об’єднання двох невеликих куп.
    - Виконання обходів для одного з біноміальних дерев у купі.
    - Демонстрацію результатів експериментального дослідження (графіки, таблиці).
3. Бути готовим відповісти на питання:
    - Яка структура біноміального дерева Bk?
    - Як біноміальна купа гарантує унікальність порядку дерев?
    - Поясніть принцип роботи операції UNION.
    - Чому INSERT та EXTRACT-MIN використовують UNION?
    - Яка роль покажчика parent у вузлі біноміального дерева?

- Опишіть амортизовану часову складність операції DECREASE-KEY.
- У яких випадках біноміальні купи є кращим вибором порівняно з бінарними купами?
- Як реалізується обхід в ширину?

7. Теми лабораторних завдань

Блок 1.

1. Базова реалізація біноміальної купи: Реалізувати MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN. Провести тестування.
2. Реалізація UNION та DELETE: Доповнити попередню роботу реалізацією UNION та DELETE. Продемонструвати їх роботу.
3. Реалізація DECREASE-KEY: Доповнити реалізацію операцією DECREASE-KEY. Продемонструвати її вплив на структуру купи.
4. Обходи біноміального дерева: Реалізувати всі три обходи (прямий, зворотний, в ширину) для окремого біноміального дерева.
5. Експериментальне дослідження INSERT та EXTRACT-MIN: Провести вимірювання часу для INSERT та EXTRACT-MIN на різних обсягах даних, порівняти з теоретичними оцінками.
6. Експериментальне дослідження UNION та DECREASE-KEY: Провести вимірювання часу для UNION та DECREASE-KEY на різних обсягах даних, порівняти з теоретичними оцінками.
7. Порівняння з бінарною купою (теоретичне та практичне): Реалізувати просту бінарну купу та порівняти її продуктивність з біноміальною купою для операцій вставки, вилучення мінімуму та, якщо можливо, об’єднання (об’єднання для бінарної купи буде O(N)).
8. Реалізація пріоритетної черги на базі біноміальної купи: Розробити інтерфейс пріоритетної черги, використовуючи біноміальну купу як базову структуру.
9. Застосування біноміальної купи в алгоритмі Дейкстри: Реалізувати алгоритм Дейкстри для знаходження найкоротших шляхів, використовуючи біноміальну купу як пріоритетну чергу.

10. Застосування біноміальної купи в алгоритмі Прима: Реалізувати алгоритм Прима для знаходження мінімального остовного дерева, використовуючи біноміальну купу.

11. Візуалізація операцій (концептуальна): Розробити текстову або просту графічну візуалізацію (наприклад, за допомогою ASCII-графіки або виведення в консоль) для демонстрації кроків операцій INSERT та UNION на невеликих прикладах.

12. Розширення функціоналу: INCREASE-KEY: Реалізувати операцію INCREASE-KEY (збільшення ключа), яка вимагатиме "бульбашкового" опускання вузла вниз або видалення та повторної вставки.

Блок 2

Завдання 1: Побудова Біноміальних Дерев Реалізуйте функцію для побудови біноміального дерева заданого порядку. Виведіть дерева B_0, B_1, B_2, B_3.

Завдання 2: Створення Біноміальної Купи Реалізуйте базові операції створення пустої купи та перевірки на порожність.

Завдання 3: Вставка Елементів Реалізуйте операцію вставки. Створіть купу з елементами [10, 5, 15, 3, 8, 12, 20].

Завдання 4: Пошук Мінімального Елемента Реалізуйте функцію пошуку мінімального елемента. Протестуйте на створеній купі.

Завдання 5: Об'єднання Куп Створіть дві купи та реалізуйте їх об'єднання. Порівняйте результат з очікуваним.

Завдання 6: Видалення Мінімального Елемента Реалізуйте операцію видалення мінімального елемента. Покажіть зміни в структурі купи.

Завдання 7: Зменшення Ключа Реалізуйте операцію зменшення ключа. Продемонструйте відновлення властивості купи.

Завдання 8: Видалення Довільного Елемента Реалізуйте операцію видалення довільного елемента через зменшення ключа.

Завдання 9: Обходи Дерев Реалізуйте всі види обходів (прямий, зворотний, симетричний, за рівнями) для біноміального дерева.

Завдання 10: Аналіз Складності Проведіть експериментальний аналіз часової складності операцій для різних розмірів куп.

Завдання 11: Візуалізація Створіть функцію для графічного представлення біноміальної купи.

Завдання 12: Алгоритм Дійкстри Реалізуйте алгоритм Дійкстри з використанням біноміальної купи як пріоритетної черги.

Тести

1. Яке з наведених тверджень є правильним щодо біноміального дерева Bk?

а) Воно має k вузлів.

б) Його висота дорівнює 2k.

в) Воно має рівно 2k вузлів.

г) Його корінь має ступінь 0.

2. Яка властивість біноміальної купи гарантує, що найменший елемент завжди знаходиться в корені одного з її біноміальних дерев?

а) Властивість унікальності порядку.

б) Властивість мінімальної купи.

в) Властивість повноти дерева.

г) Властивість збалансованості.

3. Яка операція є центральною для біноміальних куп і використовується як підпрограма в більшості інших операцій?

а) FIND-MINIMUM

б) INSERT

в) DECREASE-KEY

г) UNION

4. Яка часова складність операції INSERT у біноміальної купі в амортизованому випадку?

а) O(n)

б) O(logn)

в) O(1)

г) O(nlogn)

5. Який алгоритм на графах може ефективно використовувати біноміальну купу завдяки її операціям EXTRACT-MIN та DECREASE-KEY?

а) Сортування злиттям

б) Алгоритм Дейкстри

в) Сортування бульбашкою

г) Бінарний пошук

Тести підвищеної складності

1. Біноміальна купа містить N вузлів. Якщо N=25, які порядки біноміальних дерев будуть присутні в цій купі?

а) B0,B2,B3,B4

б) B1,B2,B4

в) B0,B3,B4

г) B0,B1,B2,B3

2. При об’єднанні двох біноміальних дерев Bk та Bk для формування Bk+1, корінь якого дерева стає дочірнім для іншого?

а) Корінь дерева з меншим ключем стає дочірнім.

б) Корінь дерева з більшим ключем стає дочірнім.

в) Це залежить від висоти дерев.

г) Це залежить від кількості вузлів у деревах.

3. Чому операція DELETE елемента в біноміальної купі часто реалізується через комбінацію DECREASE-KEY та EXTRACT-MIN?

а) Це дозволяє досягти кращої амортизованої складності.

б) Це спрощує логіку видалення довільного вузла, перетворюючи його на видалення мінімального.

в) Це єдиний спосіб зберегти властивість купи.

г) Це дозволяє уникнути рекурсії.

4. Яка з наведених операцій у біноміальної купі має найгіршу часову складність O(logn), але може бути оптимізована до O(1) за рахунок додаткових накладних витрат на підтримку?

а) UNION

б) INSERT

в) FIND-MINIMUM

г) DECREASE-KEY

5. Який з обходів N-арного дерева зазвичай використовує чергу для своєї реалізації?

а) Прямий обхід (Preorder Traversal)

б) Зворотний обхід (Postorder Traversal)

в) Обхід у ширину (Level-order Traversal)

г) Симетричний обхід (Inorder Traversal)

6. Якщо біноміальна купа складається з біноміальних дерев B0,B1,B3, скільки всього вузлів у цій купі?

а) 7

б) 8

в) 11

г) 13

7. Яка з наведених структур даних має кращу амортизовану часову складність для операції DECREASE-KEY порівняно з біноміальною купою ?

а) Бінарна купа

б) Купа Фібоначчі

в) Д-арна купа

г) Лівийська купа

Відповіді на тестові питання:

Тести

1. в)
2. б)
3. г)
4. б)
5. б)

Тести підвищеної складності

1: в) B_0, B_3, B_4. Двійкове представлення числа 25 є 11001_2, що відповідає $1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4$.

2: б) Корінь дерева з більшим ключем стає дочірнім. Це необхідно для збереження властивості мінімальної купи.

3: б) Це спрощує логіку видалення довільного вузла, перетворюючи його на видалення мінімального. Зменшення ключа до $-\infty$ гарантує, що елемент стане мінімальним і буде вилучений стандартною операцією `EXTRACT-MIN`.

4: в) FIND-MINIMUM. Хоча зазвичай сканування коренів займає $O(\log n)$, якщо підтримувати покажчик на мінімальний елемент, цю операцію можна виконати за $O(1)$ часу, додаючи незначні накладні витрати на інші операції.

5: в) Обхід у ширину (Level-order Traversal). Цей обхід систематично відвідує вузли рівень за рівнем, що ефективно реалізується за допомогою черги.

6: в) 11. Біноміальне дерево B_0 має $2^0 = 1$ вузол, B_1 має $2^1 = 2$ вузли, а B_3 має $2^3 = 8$ вузлів. Загальна кількість вузлів: $1 + 2 + 8 = 11$.

7: б) Купа Фібоначчі. Купа Фібоначчі забезпечує амортизовану часову складність $O(1)$ для операції `DECREASE-KEY`, що є кращим показником порівняно з $O(\log n)$ у біноміальної купи.

Завдання для самостійної роботи:

1. Намалюйте біноміальні дерева B_0, B_1, B_2, B_3 та B_4. Позначте кількість вузлів та висоту для кожного.
2. Побудуйте біноміальну купу, що містить 17 елементів. Вкажіть, які біноміальні дерева її складають.
3. Покроково продемонструйте операцію `INSERT(H, 15)` для біноміальної купи, що складається з $B_1(8)$ та $B_3(20)$.
4. Покроково продемонструйте операцію `EXTRACT-MIN(H)` для біноміальної купи з елементами {12,7,25,3,18,9,30}. Припустіть, що 3 є коренем B_2.
5. Поясніть, чому операція `DECREASE-KEY` вимагає покажчиків на батьківські вузли.
6. Розробіть алгоритм для перевірки, чи є задане дерево біноміальним деревом B_k.
7. Запропонуйте, як можна оптимізувати операцію `MINIMUM` у біноміальній купі, якщо вона викликається дуже часто.
8. Порівняйте амортизовану та найгіршу часову складність операцій `INSERT` та `EXTRACT-MIN` для біноміальної купи.
9. Опишіть, як можна реалізувати операцію `INCREASE-KEY` (збільшення ключа) для біноміальної купи. Яка її часова складність?
10. Дослідить, як біноміальні купи можуть бути використані в алгоритмі злиття k відсортованих списків.

Купа Фібоначчі

Купа Фібоначчі є однією з найскладніших, але водночас найефективніших структур даних для реалізації операцій з чергами з пріоритетом. Вона була розроблена Майклом Л. Фредманом та Робертом Е. Тар'яном у 1984 році та опублікована у науковому журналі у 1987 році.

Назва "Купа Фібоначчі" походить від чисел Фібоначчі, які використовуються в аналізі її часу виконання.

Купа Фібоначчі – унікальна та потужна структура даних, яка відіграє ключову роль у багатьох високоефективних алгоритмах, зокрема в алгоритмі Дейкстри та алгоритмі Прима. На відміну від звичайних бінарних або біноміальних куп, купа Фібоначчі є більш гнучкою, що дозволяє їй досягати амортизованої продуктивності, яка є кращою для деяких операцій. Це означає, що хоча окремі операції можуть займати багато часу, серія таких операцій у середньому буде виконуватися значно швидше.

Купа Фібоначчі — це набір міні-куп на основі дерев, що означає, що кожен вузол у дереві має значення, менше або рівне значенню своїх дочірніх вузлів. Звідси випливає, що мінімальний ключ у всій купі завжди знаходиться в корені одного з дерев.

На відміну від бінарних куп, які зазвичай є одним деревом, купа Фібоначчі складається з **списку кореневих вузлів** окремих дерев.

Кожен вузол у купі Фібоначчі містить наступну інформацію:

- **`ключ` (key):** Значення вузла, за яким він упорядковується в купі (наприклад, пріоритет).
- **`ступінь` (degree):** Кількість дочірніх вузлів у цього вузла. Це важливий параметр для підтримки властивостей купи.
- **`батько` (parent):** Посилання на батьківський вузол. Для кореневих вузлів це посилання буде `NULL`.
- **`дочірній_вузол` (child):** Посилання на один з дочірніх вузлів. Оскільки вузол може мати багато дочірніх вузлів, це посилання зазвичай вказує на перший дочірній вузол у круговому двозв'язному списку дочірніх вузлів.
- **`лівий_брат` (left_sibling) та `правий_брат` (right_sibling):** Посилання на лівого та правого братів у круговому двозв'язному списку, який утворюють дочірні вузли одного батька. Це дозволяє ефективно додавати та видаляти вузли з цього списку.

– **`позначений` (marked):** Булевий прапорець, який вказує, чи був вузол вже “вирізаний” зі свого батьківського списку дочірніх вузлів з моменту останнього перетворення його на кореневий вузол. Цей прапорець є ключовим для підтримки амортизованої продуктивності та запобігання надмірному зменшенню висоти дерев.

Операції Купи Фібоначчі

Купи Фібоначчі підтримують наступні операції, багато з яких мають вигідну амортизовану часову складність:

1. `Make-Fib-Heap()`: Створення порожньої Купи Фібоначчі
2. `Fib-Heap-Insert(H, x)`: Вставка нового вузла
3. `Fib-Heap-Minimum(H)`: Пошук мінімального елемента
4. `Fib-Heap-Extract-Min(H)`: Вилучення мінімального елемента
5. `Fib-Heap-Union(H1, H2)`: Об’єднання двох Куп Фібоначчі
6. `Fib-Heap-Decrease-Key(H, x, k)`: Зменшення ключа вузла
7. `Fib-Heap-Delete(H, x)`: Видалення вузла

Давайте детально розглянемо кожну з цих операцій.

Створення порожньої Купи Фібоначчі `Make-Fib-Heap()`

Ця операція створює та повертає нову, порожню купу Фібоначчі. За своєю суттю, це просто ініціалізація структури, яка буде зберігати посилання на мінімальний елемент купи та загальну кількість вузлів.

Опис: Купа Фібоначчі `H` може бути представлена як об’єкт, який має два основних поля:

– `H.min`: Посилання на вузол з найменшим ключем у купі. Це посилання вказує на один з кореневих вузлів. Якщо купа порожня, `H.min` буде `NULL`.

– `H.n`: Загальна кількість вузлів у купі. Коли купа порожня, `H.n` буде `0`.

Уявіть порожню площину. Це ваша купа Фібоначчі. На ній ще немає жодних вузлів.

Купа Фібоначчі H
H.min = NULL H.n = 0

Алгоритм:

```
Make-Fib-Heap()
  H.min = NULL  // Ініціалізуємо посилання на мінімальний вузол як NU
LL
  H.n = 0       // Встановлюємо кількість вузлів у 0
  Повернути H   // Повертаємо порожню купу
```

Псевдокод:

```
FUNCTION Make-Fib-Heap():
  H = NEW FibonacciHeapObject
  H.min = NIL
  H.n = 0
  RETURN H
```

Вставка нового вузла `Fib-Heap-Insert(H, x)`

Ця операція додає новий вузол `x` до купи Фібоначчі `H`. Вставлений вузол стає новим кореневим вузлом і додається до списку кореневих вузлів купи.

Опис: Для вставки нового вузла `x` необхідно виконати такі кроки:

1. Ініціалізація вузла: Створити новий вузол `x` з заданим ключем. Всі його поля ініціалізуються:
    - `x.degree = 0` (новий вузол не має дочірніх вузлів)
    - `x.parent = NULL` (він буде кореневим вузлом)
    - `x.child = NULL` (немає дочірніх вузлів)
    - `x.marked = FALSE` (не був вирізаний)
2. Додавання до кореневого списку: Вузол `x` додається до кругового двозв’язного списку кореневих вузлів купи `H`. Це означає, що `x.left_sibling` і `x.right_sibling` будуть вказувати на інші кореневі вузли, або на самого себе, якщо `x` є єдиним кореневим вузлом.

3. **Оновлення мінімального елемента:** Якщо `H.min` є `NULL` (купа була порожня) або ключ `x` менший за ключ поточного `H.min`, то `H.min` оновлюється до `x`.

4. **Збільшення кількості вузлів:** Збільшити `H.n` на 1.

Припустимо, у нас є купа `H` з трьома кореневими вузлами: `A`, `B`, `C`, де `B` є мінімальним. Ми хочемо вставити новий вузол `D`.

До вставки:

```
  H.min -> B

  Корінь список: A <-> B <-> C <-> A (круговий)

      A       B       C
     / \     / \     / \
    ...     ...     ...
```

Після вставки D (з ключем меншим за B):

```
  H.min -> D

  Корінь список: A <-> B <-> C <-> D <-> A (круговий)
  (Порядок може бути іншим, залежить від реалізації додавання в круго
вий список)

      A       B       C       D
     / \     / \     / \
    ...     ...     ...
```

Алгоритм:

```
Fib-Heap-Insert(H, x)
  x.degree = 0
  x.parent = NULL
  x.child = NULL
  x.marked = FALSE
  Якщо H.min == NULL
    H.min = x
    x.left_sibling = x
    x.right_sibling = x
  Інакше
    Додати x до кореневого списку H.min (наприклад, праворуч від H.mi
n)
    x.left_sibling = H.min
    x.right_sibling = H.min.right_sibling
    H.min.right_sibling.left_sibling = x
    H.min.right_sibling = x
```

```
    Якщо x.key < H.min.key
      H.min = x
  H.n = H.n + 1
```

Псевдокод такий:

```
FUNCTION Fib-Heap-Insert(H, x):
  // Ініціалізація вузла x
  x.degree = 0
  x.parent = NIL
  x.child = NIL
  x.marked = FALSE

  // Додавання x до кореневого списку H
  IF H.min IS NIL:
    H.min = x
    x.left_sibling = x
    x.right_sibling = x
  ELSE:
    // Вставляємо x між H.min та H.min.right_sibling
    x.right_sibling = H.min.right_sibling
    x.left_sibling = H.min
    H.min.right_sibling.left_sibling = x
    H.min.right_sibling = x
    IF x.key < H.min.key:
      H.min = x

  H.n = H.n + 1
```

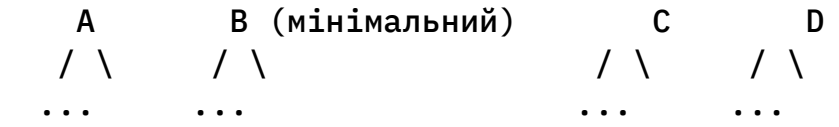
Пошук мінімального елемента `Fib-Heap-Minimum(H)`

Ця операція повертає вузол з найменшим ключем у купі Фібоначчі **`H`**. Це найпростіша операція, оскільки купа завжди підтримує пряме посилання на мінімальний елемент.

Опис: Купа Фібоначчі зберігає посилання на вузол з мінімальним ключем у полі **`H.min`**. Тому, щоб знайти мінімальний елемент, достатньо просто повернути значення цього поля. Ця операція займає константний час **`O(1)`**.

Припустимо, у нас є купа **`H`** з декількома кореневими вузлами. Посилання **`H.min`** завжди вказує на вузол з найменшим ключем.

```
  H.min -> B

  Корінь список: A <-> B <-> C <-> D <-> A
```



Алгоритм:

```
Fib-Heap-Minimum(H)
  Повернути H.min
```

Псевдокод такий:

```
FUNCTION Fib-Heap-Minimum(H):
  RETURN H.min
```

Видалення мінімального елемента `Fib-Heap-Extract-Min(H)`

Ця операція видаляє вузол з найменшим ключем з купи Фібоначчі `H` і повертає його. Це одна з найскладніших операцій, оскільки вона вимагає відновлення властивостей купи після видалення кореневого мінімального елемента.

Опис: Видалення мінімального елемента включає наступні кроки:

1. Збереження мінімального вузла: Зберегти посилання на поточний мінімальний вузол `z = H.min`.
2. Додавання дочірніх вузлів до кореневого списку: Всі дочірні вузли `z` стають кореневими вузлами купи `H`. Для кожного дочірнього вузла `x` вузла `z`:
    1. Встановити `x.parent = NULL`.
    2. Додати `x` до кореневого списку `H`.
3. Видалення `z` з кореневого списку: Видалити вузол `z` з кореневого кругового двозв’язного списку.
4. Оновлення `H.min` та консолідація:
    - Якщо `z` був єдиним вузлом у купі, `H.min` стає `NULL`.
    - Інакше, `H.min` встановлюється на довільний вузол з кореневого списку (наприклад, на `z.right_sibling`). Потім виконується операція `Consolidate(H)`. Ця операція є критично важливою для підтримки амортизованої продуктивності та зменшення кількості дерев у кореневому списку.

5. Зменшення кількості вузлів: Зменшити `H.n` на 1.
6. Повернення `z`: Повернути видалений вузол `z`.

Консолідація Кореневого Списку `Consolidate(H)`:

Операція `Consolidate(H)` є серцем амортизованої продуктивності купи Фібоначчі. Її мета — зменшити кількість кореневих вузлів, об'єднуючи дерева з однаковим ступенем. Це робиться для того, щоб кількість кореневих вузлів була логарифмічною відносно кількості вузлів у купі.

Опис `Consolidate(H)`:

1. Ініціалізація масиву `A`: Створюється допоміжний масив `A` розміром до `D(H.n)`, де `D(H.n)` — це максимальний ступінь, який може мати кореневе дерево у купі Фібоначчі з `H.n` вузлами. Цей розмір приблизно дорівнює `log_phi(H.n)`, де `phi` — золотий перетин. Всі елементи `A` ініціалізуються як `NULL`.
2. Перебір кореневих вузлів: Пройти по всіх кореневих вузлах у круговому двозв'язному списку купи `H`. Для кожного кореневого вузла `x`:
    - Зберегти його ступінь `d = x.degree`.
    - Об'єднання дерев: Поки `A[d]` не `NULL` (тобто вже існує інше дерево з таким же ступенем):
        - Нехай `y = A[d]`.
        - Визначити, який з `x` і `y` має менший ключ. Нехай `min_node` буде вузлом з меншим ключем, а `max_node` — вузлом з більшим ключем.
        - `Fib-Heap-Link(H, max_node, min_node)`: Видалити `max_node` з кореневого списку та зробити його дочірнім вузлом `min_node`. Збільшити `min_node.degree` на 1. Встановити `max_node.parent = min_node` та `max_node.marked = FALSE`.
        - Встановити `A[d] = NULL`.
        - Збільшити `d` на 1, оскільки `min_node` тепер має більший ступінь.

- Встановити `A[d] = x`. (Вузол `x` (або новоутворений об'єднаний вузол) тепер є єдиним деревом зі ступенем `d`).

3. Відновлення `H.min` та створення нового кореневого списку: Після обробки всіх кореневих вузлів, пройти по масиву `A`. Для кожного не `NULL` елемента `A[i]`:

   - Додати `A[i]` до нового (або оновленого) кореневого списку купи `H`.
   - Оновити `H.min`, якщо `A[i].key` менший за поточний `H.min.key`.
   - Очистити `H.min` і потім додати всі елементи з `A` до кореневого списку, оновлюючи `H.min` відповідно. Це забезпечує, що `H.min` вказує на справжній мінімум після консолідації.

`Fib-Heap-Link(H, y, x)`: Ця допоміжна функція робить вузол `y` дочірнім вузлом `x`. Вона видаляє `y` з кореневого списку та додає його до списку дочірніх вузлів `x`.

Розглянемо виконання `Extract-Min`:

Нехай до `Extract-Min`:

```
  H.min -> (B, key=5)

  Корінь список:
(A, 10) <-> (B, 5) <-> (C, 15) <-> (D, 7) <-> (A, 10)

      (A, 10)     (B, 5)       (C, 15)     (D, 7)
       /  \        /  \         /  \        /  \
      (E, 20)    (F, 8) (G, 12)  ...         ...
                  / \
               (H, 9) (I, 11)
```

Крок 1: Зберегти B, додати дочірні F, G до кореневого списку, видалити B.

```
  H.min -> (A, 10) (тимчасово, потім буде перераховано)

  Корінь список:
(A, 10) <-> (C, 15) <-> (D, 7) <-> (F, 8) <-> (G, 12) <-> (A, 10)

      (A, 10)     (C, 15)     (D, 7)     (F, 8)      (G, 12)
       /  \        /  \        /  \       /  \
      (E, 20)      ...         ...     (H, 9) (I, 11)
```

Крок 2: Консолідація.

Припустимо, після обробки, ми об’єднаємо дерева, щоб зменшити кількість кореневих вузлів. Наприклад, якщо A і D мають однаковий ступінь, вони можуть бути об’єднані.

Після `Consolidate`:

```
  H.min -> (D, 7) (припустимо, що D стане мінімальним після консоліда
ції та об'єднання)

  Корінь список:
(J, new_key) <-> (D, 7) <-> (K, new_key) <-> (J, new_key)

      (J, new_key)      (D, 7) (мінімальний)     (K, new_key)
       /   \             /   \                     /   \
        ...               ...                       ...
```

Алгоритм `Fib-Heap-Extract-Min(H)`:

```
Fib-Heap-Extract-Min(H)
  z = H.min
  Якщо z != NULL
    // Додати дочірні вузли z до кореневого списку
    Якщо z.child != NULL
      curr_child = z.child
      повторювати
        next_child = curr_child.right_sibling
        curr_child.parent = NULL
        // Додати curr_child до кореневого списку H
        curr_child.right_sibling = H.min.right_sibling
        curr_child.left_sibling = H.min
        H.min.right_sibling.left_sibling = curr_child
        H.min.right_sibling = curr_child
        curr_child = next_child
      поки curr_child != z.child

    // Видалити z з кореневого списку
    z.left_sibling.right_sibling = z.right_sibling
    z.right_sibling.left_sibling = z.left_sibling

    Якщо z == z.right_sibling // z був єдиним кореневим вузлом
      H.min = NULL
    Інакше
      H.min = z.right_sibling // Встановлюємо тимчасово H.min
      Consolidate(H) // Викликаємо консолідацію
    H.n = H.n - 1
  Повернути z
```

Алгоритм `Consolidate(H)`:

```
Consolidate(H)
  D_max = log_phi(H.n) // Приблизний максимальний ступінь, phi = (1 + sqrt(5)) / 2
  A = масив розміру D_max + 1, ініціалізований NULL

  // Зібрати всі кореневі вузли
  root_list = []
  if H.min != NULL:
    current = H.min
    start = H.min
    repeat:
      root_list.append(current)
      current = current.right_sibling
    while current != start

  // Пройти по кожному кореневому вузлу
  для кожного w в root_list:
    x = w
    d = x.degree
    поки A[d] != NULL:
      y = A[d] // Інший вузол з тим же ступенем
      Якщо x.key > y.key: // Переконаємось, що x є вузлом з меншим ключем
        temp = x
        x = y
        y = temp
      Fib-Heap-Link(H, y, x) // Зробити y дочірнім для x
      A[d] = NULL
      d = d + 1
    A[d] = x

  H.min = NULL // Очистити H.min для перерахунку

  // Відбудувати кореневий список та знайти новий мінімум
  для i від 0 до D_max:
    Якщо A[i] != NULL:
      Якщо H.min == NULL:
        H.min = A[i]
        A[i].left_sibling = A[i]
        A[i].right_sibling = A[i]
      Інакше:
        // Додати A[i] до кореневого списку H.min
        A[i].right_sibling = H.min.right_sibling
```

```
        A[i].left_sibling = H.min
        H.min.right_sibling.left_sibling = A[i]
        H.min.right_sibling = A[i]
        Якщо A[i].key < H.min.key:
          H.min = A[i]
```

Алгоритм `Fib-Heap-Link(H, y, x)`:

```
Fib-Heap-Link(H, y, x)
  // Видалити y з кореневого списку
  y.left_sibling.right_sibling = y.right_sibling
  y.right_sibling.left_sibling = y.left_sibling

  // Зробити y дочірнім для x
  Якщо x.child == NULL:
    x.child = y
    y.right_sibling = y
    y.left_sibling = y
  Інакше:
    // Додати y до кругового двозв'язного списку дочірніх вузлів x
    y.right_sibling = x.child.right_sibling
    y.left_sibling = x.child
    x.child.right_sibling.left_sibling = y
    x.child.right_sibling = y
  y.parent = x
  x.degree = x.degree + 1
  y.marked = FALSE // Обов'язково знімаємо позначку при приєднанні
```

Псевдокод `Fib-Heap-Extract-Min(H)`:

```
FUNCTION Fib-Heap-Extract-Min(H):
  z = H.min
  IF z IS NOT NIL:
    // Перемістити дочірні елементи z до кореневого списку
    IF z.child IS NOT NIL:
      current_child = z.child
      DO:
        next_child = current_child.right_sibling
        current_child.parent = NIL
        // Додати current_child до кореневого списку H
        current_child.right_sibling = H.min.right_sibling
        current_child.left_sibling = H.min
        H.min.right_sibling.left_sibling = current_child
        H.min.right_sibling = current_child
        current_child = next_child
```

```
    WHILE current_child IS NOT z.child

  // Видалити z з кореневого списку
  z.left_sibling.right_sibling = z.right_sibling
  z.right_sibling.left_sibling = z.left_sibling

  IF z IS z.right_sibling: // z був єдиним вузлом
    H.min = NIL
  ELSE:
    H.min = z.right_sibling
    Consolidate(H) // Викликати консолідацію
  H.n = H.n - 1
RETURN z
```

Псевдокод `Consolidate(H)`:

```
FUNCTION Consolidate(H):
  D_max = CEILING(LOG_PHI(H.n)) // LOG_PHI(N) де PHI = (1 + SQRT(5))
/2
  A = ARRAY OF SIZE (D_max + 1) INITIALIZED TO NIL

  // Зібрати всі кореневі вузли
  root_nodes_to_process = LIST()
  IF H.min IS NOT NIL:
    current = H.min
    start_node = H.min
    REPEAT:
      root_nodes_to_process.ADD(current)
      current = current.right_sibling
    UNTIL current IS start_node

  FOR EACH w IN root_nodes_to_process:
    x = w
    d = x.degree
    WHILE A[d] IS NOT NIL:
      y = A[d]
      IF x.key > y.key:
        SWAP(x, y) // Ensure x has the smaller key
      Fib-Heap-Link(H, y, x)
      A[d] = NIL
      d = d + 1
    A[d] = x

  H.min = NIL // Очистити H.min для нового визначення
```

```
  // Відбудувати кореневий список з A та знайти новий мінімум
  FOR i FROM 0 TO D_max:
    IF A[i] IS NOT NIL:
      IF H.min IS NIL:
        H.min = A[i]
        A[i].left_sibling = A[i]
        A[i].right_sibling = A[i]
      ELSE:
        // Додати A[i] до кругового списку H.min
        A[i].right_sibling = H.min.right_sibling
        A[i].left_sibling = H.min
        H.min.right_sibling.left_sibling = A[i]
        H.min.right_sibling = A[i]
        IF A[i].key < H.min.key:
          H.min = A[i]
```

Псевдокод `Fib-Heap-Link(H, y, x)` такий:

```
FUNCTION Fib-Heap-Link(H, y, x):
  // Видалити y з кореневого списку
  y.left_sibling.right_sibling = y.right_sibling
  y.right_sibling.left_sibling = y.left_sibling

  // Зробити y дочірнім для x
  IF x.child IS NIL:
    x.child = y
    y.right_sibling = y
    y.left_sibling = y
  ELSE:
    // Додати y до кругового двозв'язного списку дочірніх вузлів x
    y.right_sibling = x.child.right_sibling
    y.left_sibling = x.child
    x.child.right_sibling.left_sibling = y
    x.child.right_sibling = y
  y.parent = x
  x.degree = x.degree + 1
  y.marked = FALSE // Знімаємо позначку, так як y стає дочірнім
```

Об’єднання двох Куп Фібоначчі `Fib-Heap-Union(H1, H2)`

Ця операція об’єднує дві купи Фібоначчі `H1` і `H2` в одну нову купу. Вона є дуже ефективною, оскільки по суті полягає лише в об’єднанні кореневих списків і виборі нового мінімального елемента.

Опис:

1. Створення нової купи: Створити нову порожню купу **H_new**.
2. Об’єднання кореневих списків: Об’єднати кругові двозв’язні списки кореневих вузлів **H1** та **H2**. Це робиться шляхом “зшивання” їхніх списків. Наприклад, можна взяти **H1.min** і **H2.min** і з’єднати їхні **right_sibling** та **left_sibling** посилання таким чином, щоб утворився один великий круговий список.
3. Визначення нового мінімуму: **H_new.min** встановлюється на **H1.min**. Якщо **H2.min** існує і його ключ менший за ключ **H1.min**, то **H_new.min** оновлюється до **H2.min**.
4. Оновлення кількості вузлів: **H_new.n** встановлюється як сума **H1.n + H2.n**.
5. Очищення старих куп: **H1** і **H2** стають порожніми (або їхні посилання на мінімум і кількість вузлів обнуляються).

До об’єднання:

```
H1.min -> A                        H2.min -> E

Корінь список H1: A <-> B <-> C <-> A
Корінь список H2: E <-> F <-> G <-> E

    A       B       C           E       F       G
   / \     / \     / \         / \     / \     / \
   ...     ...     ...         ...     ...     ...
```

Після об’єднання (припустимо, A < E):

```
H_new.min -> A

Корінь список H_new:
A <-> B <-> C <-> E <-> F <-> G <-> A (круговий)

    A       B       C       E       F       G
   / \     / \     / \     / \     / \     / \
  ...     ...     ...     ...     ...     ...
```

Алгоритм:

```
Fib-Heap-Union(H1, H2)
  H_new = Make-Fib-Heap() // Створюємо нову порожню купу
```

```
  // Об'єднуємо кореневі списки H1 і H2
  // Припускаємо, що H1.min і H2.min існують
  // Це спрощена версія. Реальна вимагає обробки порожніх куп.
  Якщо H1.min == NULL:
    H_new.min = H2.min
  Інакше Якщо H2.min == NULL:
    H_new.min = H1.min
  Інакше:
    H_new.min = H1.min
    // З'єднуємо списки:
    // Зберігаємо посилання на наступні елементи після мінімумів
    H1_min_right = H1.min.right_sibling
    H2_min_right = H2.min.right_sibling

    // З'єднуємо H1.min з H2.min
    H1.min.right_sibling = H2.min
    H2.min.left_sibling = H1.min

    // З'єднуємо попередній кінець H2 з попереднім початком H1
    H2_min_right.left_sibling = H1_min_right.left_sibling
    H1_min_right.left_sibling.right_sibling = H2_min_right

    // Оновлюємо мінімум, якщо H2.min менший
    Якщо H2.min.key < H1.min.key:
      H_new.min = H2.min

  H_new.n = H1.n + H2.n
  Повернути H_new
```

Псевдокод такий:

```
FUNCTION Fib-Heap-Union(H1, H2):
  H_new = Make-Fib-Heap()

  // Об'єднання кореневих списків
  IF H1.min IS NIL:
    H_new.min = H2.min
  ELSE IF H2.min IS NIL:
    H_new.min = H1.min
  ELSE:
    H_new.min = H1.min

    // З'єднуємо списки H1 та H2
    // Зберігаємо посилання на "крайні" елементи
    H1_end = H1.min.left_sibling
```

```
    H2_end = H2.min.left_sibling

    // З'єднуємо H1.min.left_sibling з H2.min
    H1_end.right_sibling = H2.min
    H2.min.left_sibling = H1_end

    // З'єднуємо H2.min.left_sibling з H1.min
    H2_end.right_sibling = H1.min
    H1.min.left_sibling = H2_end

    // Оновлюємо H_new.min якщо H2.min є меншим
    IF H2.min.key < H1.min.key:
      H_new.min = H2.min

  H_new.n = H1.n + H2.n

  // Очищення старих куп (їхня внутрішня структура вже змінена)
  // Це не обов'язково для коректності операції, але корисно для упра
вління пам'яттю
  H1.min = NIL
  H1.n = 0
  H2.min = NIL
  H2.n = 0

  RETURN H_new
```

Зменшення ключа вузла `Fib-Heap-Decrease-Key(H, x, k)`

Ця операція зменшує ключ вузла `x` до нового значення `k`. Важливо, що `k` повинно бути меншим за поточний ключ `x.key`. Якщо після зменшення ключа властивість міні-купи порушується (тобто `x.key < x.parent.key`), то вузол `x` “вирізається” зі свого батьківського дерева і стає кореневим вузлом. Цей процес може викликати каскадні вирізання.

Опис:

1. Перевірка вхідних даних: Переконатися, що новий ключ `k` дійсно менший за поточний ключ `x.key`. Якщо ні, це є помилкою.
2. Оновлення ключа: Встановити `x.key = k`.
3. Перевірка порушення властивості купи: Якщо `x.parent` існує і `x.key < x.parent.key`:

- Викликати `Cut(H, x, x.parent)`. Ця операція “вирізає” `x` з його батьківського дерева.
- Викликати `Cascading-Cut(H, x.parent)`. Ця операція перевіряє, чи не порушилася властивість позначки (`marked`) у батьківського вузла після вирізання його дочірнього елемента.

4. Оновлення `H.min`: Якщо `x.key` менший за ключ `H.min`, то `H.min` оновлюється до `x`.

Видалення вузла з дочірнього списку `Cut(H, x, y)`

Ця функція видаляє вузол `x` з дочірнього списку його батька `y` та додає `x` до кореневого списку купи `H`.

Опис `Cut(H, x, y)`:

1. Видалення `x` з дочірнього списку `y`:
   - Видалити `x` з кругового двозв’язного списку дочірніх вузлів `y`.
   - Зменшити `y.degree` на 1.
   - Якщо `x` був єдиним дочірнім вузлом `y`, то `y.child` стає `NULL`. Інакше, `y.child` оновлюється до `x.right_sibling` (якщо `y.child` вказував на `x`).
2. Додавання `x` до кореневого списку: Додати `x` до кругового двозв’язного списку кореневих вузлів `H`.
3. Оновлення полів `x`: Встановити `x.parent = NULL` та `x.marked = FALSE`.

Перевірка існування позначки вузла `Cascading-Cut(H, y)`

Ця функція застосовується до батьківського вузла `y` після того, як його дочірній вузол `x` був вирізаний. Вона перевіряє, чи не був `y` вже “позначений” (marked).

Опис `Cascading-Cut(H, y)`:

1. Перевірка батька: Якщо `y.parent` не `NULL` (тобто `y` не є кореневим вузлом):
   - Якщо `y.marked` є `FALSE` (вузол `y` не був позначений):
     - Встановити `y.marked = TRUE`.

- Інакше (`y.marked` є `TRUE`):
  - Зберегти посилання на `z = y.parent`.
  - Викликати `Cut(H, y, z)` (вирізати `y` з його батька `z`).
  - Викликати `Cascading-Cut(H, z)` (рекурсивно продовжити процес для батька `z`).

Розглянемо виконання `Decrease-Key`:

Нехай до виконання `Decrease-Key(H, G, 6)` (G має ключ 12, батько F):

```
H.min -> B

Корінь список: A <-> B <-> C <-> A

    B (5)
     / \
    F (8) G (12)
     / \
    H (9) I (11)
```

Після `Decrease-Key(H, G, 6)`:

`G.key` стає `6`. Тепер `G.key (6) < F.key (8)`. Властивість купи порушена.

1. `Cut(H, G, F)`: `G` видаляється з дочірніх вузлів `F` і стає кореневим. `F.degree` зменшується. `G.parent = NULL`, `G.marked = FALSE`.
2. `Cascading-Cut(H, F)`: `F` не є кореневим і `F.marked` є `FALSE`. Отже, `F.marked` стає `TRUE`.

Після `Cut` та `Cascading-Cut` отримаємо:

```
H.min -> B (або G, якщо 6 < 5, що не є в цьому прикладі)

Корінь список: A <-> B <-> C <-> G(6) <-> A (круговий)

    B (5)        G (6)
     /
    F (8)
     / \
    H (9) I (11)
```

Алгоритм `Fib-Heap-Decrease-Key(H, x, k)`:

```
Fib-Heap-Decrease-Key(H, x, k)
  Якщо k > x.key
```

```
    Помилка "Новий ключ більший за поточний"
  x.key = k
  y = x.parent
  Якщо y != NULL і x.key < y.key // Порушення властивості міні-купи
    Cut(H, x, y)
    Cascading-Cut(H, y)
  Якщо x.key < H.min.key
    H.min = x
```

Алгоритм `Cut(H, x, y)`:

```
Cut(H, x, y)
  // Видалити x з дочірнього списку y
  Якщо x == x.right_sibling // x був єдиним дочірнім вузлом y
    y.child = NULL
  Інакше
    x.left_sibling.right_sibling = x.right_sibling
    x.right_sibling.left_sibling = x.left_sibling
    Якщо y.child == x // y.child вказував на x, оновити його
      y.child = x.right_sibling
  y.degree = y.degree - 1

  // Додати x до кореневого списку H
  x.right_sibling = H.min.right_sibling
  x.left_sibling = H.min
  H.min.right_sibling.left_sibling = x
  H.min.right_sibling = x

  x.parent = NULL
  x.marked = FALSE // Знімаємо позначку, так як x стає кореневим
```

Алгоритм `Cascading-Cut(H, y)`:

```
Cascading-Cut(H, y)
  z = y.parent
  Якщо z != NULL
    Якщо y.marked == FALSE
      y.marked = TRUE
    Інакше
      Cut(H, y, z)
      Cascading-Cut(H, z)
```

Псевдокод `Fib-Heap-Decrease-Key(H, x, k)` такий:

```
FUNCTION Fib-Heap-Decrease-Key(H, x, k):
  IF k > x.key:
    ERROR "New key is greater than current key."

  x.key = k
  y = x.parent

  IF y IS NOT NIL AND x.key < y.key: // Властивість купи порушена
    Cut(H, x, y)
    Cascading-Cut(H, y)

  IF x.key < H.min.key:
    H.min = x
```

Псевдокод `Cut(H, x, y)` такий:

```
FUNCTION Cut(H, x, y):
  // Видалити x з дочірнього списку y
  IF x IS x.right_sibling: // x був єдиним дочірнім елементом y
    y.child = NIL
  ELSE:
    x.left_sibling.right_sibling = x.right_sibling
    x.right_sibling.left_sibling = x.left_sibling
    IF y.child IS x: // Оновити y.child, якщо він вказував на x
      y.child = x.right_sibling
  y.degree = y.degree - 1

  // Додати x до кореневого списку H
  x.right_sibling = H.min.right_sibling
  x.left_sibling = H.min
  H.min.right_sibling.left_sibling = x
  H.min.right_sibling = x

  x.parent = NIL
  x.marked = FALSE // Знімаємо позначку, так як x стає кореневим вузл
ом
```

Псевдокод `Cascading-Cut(H, y)` такий:

```
FUNCTION Cascading-Cut(H, y):
  z = y.parent
  IF z IS NOT NIL:
    IF y.marked IS FALSE:
      y.marked = TRUE
    ELSE:
```

```
Cut(H, y, z)
Cascading-Cut(H, z)
```

Видалення вузла `Fib-Heap-Delete(H, x)`

Ця операція видаляє довільний вузол **x** з купи Фібоначчі **H**. Вона реалізується шляхом зменшення ключа **x** до негативної нескінченності (або дуже малого значення), що гарантує, що **x** стане мінімальним елементом, а потім викликом **Extract-Min**.

Опис:

1. Зменшення ключа: Викликати **Fib-Heap-Decrease-Key(H, x, -infinity)** (або дуже мале число, наприклад, **MIN_INT**). Це перемістить **x** до кореневого списку (можливо, з каскадними вирізаннями) і зробить його новим мінімальним елементом купи.
2. Вилучення мінімуму: Викликати **Fib-Heap-Extract-Min(H)**. Це видалить **x** (який тепер є мінімальним елементом) з купи.

До **Delete(H, G)** (G має ключ 12):

```
H.min -> B

Корінь список: A <-> B <-> C <-> A

    B (5)
     / \
    F (8) G (12)
     / \
    H (9) I (11)
```

Крок 1: **Decrease-Key(H, G, -infinity)**

G стає кореневим вузлом з дуже малим ключем і **H.min** вказує на **G**.

```
H.min -> G (-infinity)

Корінь список: A <-> B <-> C <-> G(-infinity) <-> A

    B (5)        G (-infinity)
     /
    F (8)
```

```
 / \
H (9) I (11)
```

Крок 2: `Extract-Min(H)`

`G` (який тепер є мінімальним) видаляється, його дочірні вузли стають кореневими, і запускається консолідація.

```
H.min -> B (після консолідації)

Корінь список: A <-> B <-> C <-> H(9) <-> I(11) <-> A (круговий)

    B (5)        H (9)       I (11)
     /
    F (8)
```

Алгоритм:

```
Fib-Heap-Delete(H, x)
  Fib-Heap-Decrease-Key(H, x, -infinity) // Зменшуємо ключ до найменш
ого можливого значення
  Fib-Heap-Extract-Min(H) // Видаляємо теперішній мінімальний елемент
(який є x)
```

Псевдокод такий:

```
FUNCTION Fib-Heap-Delete(H, x):
  // Припускаємо, що є константа NEGATIVE_INFINITY
  Fib-Heap-Decrease-Key(H, x, NEGATIVE_INFINITY)
  Fib-Heap-Extract-Min(H)
```

Обходи Дерева

Хоча купи Фібоначчі не є бінарними деревами і не мають строгої ієрархічної структури, як, наприклад, червоно-чорні дерева, концепція обходу все ж може бути застосована до їхніх компонентів — окремих дерев у кореневому списку. Оскільки кожне дерево в купі Фібоначчі є простою міні-купою , ми можемо застосувати до них стандартні алгоритми обходу дерев. Найбільш поширеними є:

- Прямий обхід (Pre-order traversal): Корінь, Ліве піддерево, Праве піддерево.
- Зворотний обхід (Post-order traversal): Ліве піддерево, Праве піддерево, Корінь.

– Центрований обхід (In-order traversal): Ліве піддерево, Корінь, Праве піддерево. (Застосовується до бінарних дерев, але може бути адаптований для загальних дерев).

– Обхід у ширину (Breadth-First Search, BFS): Обхід по рівнях, починаючи від кореня.

Для купи Фібоначчі, яка складається з кількох дерев, обхід зазвичай означає обхід кожного окремого дерева.

Розглянемо обхід для одного дерева, оскільки це застосовується до кожного дерева в кореневому списку купи Фібоначчі.

Структура вузла для обходу

Для зручності обходу, припустимо, що у вузла, окрім `key`, `parent`, `degree`, `marked`, є посилання на першого дочірнього вузла (`child`) та посилання на його братів (`left_sibling`, `right_sibling`). Дочірні вузли утворюють круговий двозв'язний список, тому для обходу їх потрібно ітерувати.

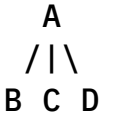
```
Структура Вузла:
  key: int
  degree: int
  parent: Node*
  child: Node* // Посилання на будь-якого дочірнього вузла
  left_sibling: Node*
  right_sibling: Node*
  marked: boolean
```

Обхід у глибину (DFS)

DFS для загальних дерев зазвичай включає Pre-order та Post-order обходи. In-order обхід є специфічним для бінарних дерев, але ми можемо адаптувати його для ітерації по дочірніх вузлах.

1.1. Прямий Обхід (Pre-order)

Відвідати кореневий вузол, потім рекурсивно відвідати всі його дочірні піддерева (в довільному порядку).



```
Порядок обходу: A -> B -> C -> D
```

Алгоритм:

```
Pre-order-Traversal(node)
  Якщо node == NULL:
    Повернути
  Відвідати(node) // Наприклад, вивести node.key

  Якщо node.child != NULL:
    first_child = node.child
    current_child = first_child
    Повторювати:
      Pre-order-Traversal(current_child)
      current_child = current_child.right_sibling
    Поки current_child != first_child
```

Псевдокод такий:

```
PROCEDURE Pre-order-Traversal(node):
  IF node IS NIL:
    RETURN

  VISIT(node) // Наприклад, PRINT node.key

  IF node.child IS NOT NIL:
    first_child = node.child
    current_child = first_child
    DO:
      Pre-order-Traversal(current_child)
      current_child = current_child.right_sibling
    WHILE current_child IS NOT first_child
```

1.2. **Зворотний Обхід** (Post-order Traversal)

Рекурсивно відвідати всі дочірні піддерева, а потім відвідати кореневий вузол.

```
Порядок обходу: B -> C -> D -> A
```

Алгоритм:

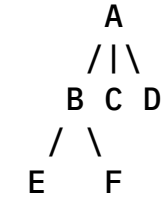
```
Post-order-Traversal(node)
  Якщо node == NULL:
    Повернути

  Якщо node.child != NULL:
    first_child = node.child
    current_child = first_child
    Повторювати:
      Post-order-Traversal(current_child)
      current_child = current_child.right_sibling
    Поки current_child != first_child

  Відвідати(node) // Наприклад, вивести node.key
```

Псевдокод такий:

```
PROCEDURE Post-order-Traversal(node):
  IF node IS NIL:
    RETURN

  IF node.child IS NOT NIL:
    first_child = node.child
    current_child = first_child
    DO:
      Post-order-Traversal(current_child)
      current_child = current_child.right_sibling
    WHILE current_child IS NOT first_child

  VISIT(node) // Наприклад, PRINT node.key
```

1.3. **Обхід у ширину** (Breadth-First Search - BFS)

Відвідати вузли рівень за рівнем, починаючи від кореня. Використовує чергу для відстеження вузлів, які потрібно відвідати.



```
Порядок обходу: A -> B -> C -> D -> E -> F
```

Алгоритм:

```
BFS-Traversal(root_node)
  Якщо root_node == NULL:
    Повернути

  queue = Нова_Черга()
  queue.Додати(root_node)

  Поки queue не порожня:
    current_node = queue.Видалити()
    Відвідати(current_node) // Наприклад, вивести current_node.key

    Якщо current_node.child != NULL:
      first_child = current_node.child
      child_to_add = first_child
      Повторювати:
        queue.Додати(child_to_add)
        child_to_add = child_to_add.right_sibling
      Поки child_to_add != first_child
```

Псевдокод такий:

```
PROCEDURE BFS-Traversal(root_node):
  IF root_node IS NIL:
    RETURN

  queue = NEW QUEUE()
  queue.ENQUEUE(root_node)

  WHILE NOT queue.IS_EMPTY():
    current_node = queue.DEQUEUE()
    VISIT(current_node) // Наприклад, PRINT current_node.key

    IF current_node.child IS NOT NIL:
      first_child = current_node.child
      child_to_add = first_child
      DO:
        queue.ENQUEUE(child_to_add)
        child_to_add = child_to_add.right_sibling
      WHILE child_to_add IS NOT first_child
```

Обхід Купи Фібоначчі Загалом

Щоб обійти всю купу Фібоначчі, потрібно застосувати один із вищезазначених обходів до кожного кореневого дерева в кореневому списку.

Алгоритм обходу всієї купи Фібоначчі

Розглянемо Прямий обхід як приклад:

```
Fib-Heap-Full-Traversal(H)
  Якщо H.min == NULL:
    Повернути

  current_root = H.min
  Повторювати:
    Pre-order-Traversal(current_root) // Або Post-order-Traversal, аб
о BFS-Traversal
    current_root = current_root.right_sibling
  Поки current_root != H.min
```

Псевдокод такий:

```
PROCEDURE Fib-Heap-Full-Traversal(H):
  IF H.min IS NIL:
    RETURN

  current_root = H.min
  start_root = H.min
  REPEAT:
    Pre-order-Traversal(current_root) // Або Post-order-Traversal(cur
rent_root), або BFS-Traversal(current_root)
    current_root = current_root.right_sibling
  UNTIL current_root IS start_root
```

Ці обходи дозволяють нам досліджувати структуру та вміст купи Фібоначчі. Хоча вони не є основними операціями купи, вони корисні для відладки, візуалізації або специфічних завдань, що вимагають перегляду всіх елементів.

Область застосування куп Фібоначчі

Купи Фібоначчі, завдяки своїй унікальній структурі та амортизованій часовій складності для багатьох операцій, знайшли своє застосування у високоефективних алгоритмах, де важлива швидкість виконання операцій зменшення ключа та об’єднання куп. Хоча вони складніші в реалізації, ніж

бінарні або біноміальні купи, їхня продуктивність стає критично важливою у певних сценаріях.

Основними областями застосування куп Фібоначчі є:

- Алгоритми на графах: Це, мабуть, найпоширеніша і найважливіша область застосування. Ефективність куп Фібоначчі значно покращує продуктивність деяких класичних алгоритмів на графах, особливо тих, що використовують операції “зменшення ключа” та “вилучення мінімуму”.
- Оптимізаційні задачі: У задачах, де необхідно динамічно оновлювати пріоритети елементів та знаходити мінімальний елемент.
- Мережеві алгоритми: У маршрутизації та інших мережевих задачах, де важлива швидка обробка пріоритетів.
- Інші спеціалізовані алгоритми: У деяких складніших алгоритмах, які потребують високоефективних пріоритетних черг.

Приклади застосування куп Фібоначчі

Давайте детальніше розглянемо конкретні приклади алгоритмів, де купи Фібоначчі виявляють свої переваги.

1. Алгоритм Дейкстри для пошуку найкоротшого шляху

Алгоритм Дейкстри знаходить найкоротші шляхи від одного джерела до всіх інших вершин у графі з невід’ємними вагами ребер. У своїй стандартній реалізації він використовує пріоритетну чергу для ефективного вибору наступної вершини для обробки.

Як застосовуються купи Фібоначчі: Під час роботи алгоритму Дейкстри, коли знаходиться коротший шлях до вже відкритої вершини, необхідно зменшити ключ (пріоритет) цієї вершини в пріоритетній черзі.

Операція `Decrease-Key` в купі Фібоначчі має амортизовану часову складність $O(1)$, що є значно кращим за $O(\log V)$ для бінарної купи.

- Крок 1: Ініціалізація. Для кожної вершини v встановлюється відстань $d[v] = \infty$, крім початкової вершини s, для якої $d[s] = 0$. Усі вершини додаються до купи Фібоначчі, де ключем є поточна відома відстань до вершини.

– Крок 2: Ітерації. Доки купа не порожня:

  - Вилучення мінімуму: Вилучається вершина u з найменшою відстанню (`Extract-Min`), яка тепер вважається остаточно обробленою.
  - Оновлення сусідів: Для кожного сусіда v вершини u: якщо $d[u] + \text{вага}(u, v) < d[v]$, то $d[v]$ оновлюється до $d[u] + \text{вага}(u, v)$, і викликається операція `Decrease-Key` для v у купі Фібоначчі.

Перевага: Використання купи Фібоначчі зменшує загальний час виконання алгоритму Дейкстри з $O(E\log V)$ (з бінарною купою) до $O(E + V\log V)$, де V — кількість вершин, E — кількість ребер. Це особливо відчутно для розріджених графів, де E значно менше, ніж V^2.

2. Алгоритм Прима для побудови мінімального остовного дерева

Опис: Алгоритм Прима знаходить мінімальне остовне дерево (MST) у зв'язаному зваженому неорієнтованому графі. Він також використовує пріоритетну чергу для вибору наступного ребра, яке слід додати до MST.

Подібно до алгоритму Дейкстри, алгоритм Прима потребує ефективного оновлення ключів (вартостей) вершин.

– **Крок 1**: Ініціалізація. Для кожної вершини v встановлюється ключ $key[v] = \infty$ (що представляє вагу найдешевшого ребра, що з'єднує v з деревом MST, що будується), крім початкової вершини r, для якої $key[r] = 0$. Усі вершини додаються до купи Фібоначчі з ключами $key[v]$.

– **Крок 2**: Ітерації. Доки купа не порожня:

  - Вилучення мінімуму: Вилучається вершина u з найменшим ключем (`Extract-Min`), яка тепер додається до MST.
  - Оновлення сусідів: Для кожного сусіда v вершини u: якщо v ще не в MST і вага ребра (u, v) менша за $key[v]$, то $key[v]$ оновлюється до ваги (u, v), і викликається операція `Decrease-Key` для v у купі Фібоначчі.

Використання купи Фібоначчі зменшує загальний час виконання алгоритму Прима з $O(E\log V)$ до $O(E + V\log V)$.

3. Алгоритм Костолома

Деякі складніші алгоритми пошуку MST для загальних графів також можуть використовувати купи Фібоначчі для покращення продуктивності, наприклад, алгоритм Борувки-Прима або алгоритм, що базується на матричному множенні.

4. Задачі Симуляції Дискретних Подій

У симуляціях, де події відбуваються в певний час і необхідно завжди обробляти подію, що настає найраніше. Пріоритетна черга може зберігати майбутні події, а операція `Decrease-Key` може використовуватися, якщо час події змінюється.

5. Задачі Мережевого Потоку

У деяких алгоритмах, пов'язаних з максимальним потоком або мінімальним розрізом у мережах, де потрібно ефективно керувати пріоритетами вузлів або ребер.

Переваги та Недоліки

Купи Фібоначчі, як і будь-яка структура даних, мають свої сильні та слабкі сторони, які визначають їхню доцільність використання в конкретних задачах.

Переваги

1. Амортизована ефективність операцій: Це головна перевага.
    - `Fib-Heap-Insert`: $O(1)$
    - `Fib-Heap-Minimum`: $O(1)$
    - `Fib-Heap-Union`: $O(1)$
    - `Fib-Heap-Decrease-Key`: $O(1)$
    - `Fib-Heap-Extract-Min`: $O(\log n)$ (де n — кількість елементів у купі). Ця амортизована продуктивність означає, що середній час виконання операції протягом довгої послідовності операцій є низьким, навіть якщо окремі операції (як `Extract-Min`) можуть займати більше часу. Це робить їх ідеальними для алгоритмів, де часто виконуються операції `Decrease-Key`.
2. Висока продуктивність для деяких алгоритмів на графах: Як вже згадувалося, вони дозволяють досягти теоретично кращої часової

складності для алгоритмів Дейкстри та Прима порівняно з бінарними купами.

3. Гнучка структура: На відміну від біноміальних куп, де дерева мають жорстко визначені ступені, купи Фібоначчі мають більш гнучку структуру, яка дозволяє відкладати деякі “дорогі” операції (наприклад, консолідацію) до моменту, коли вони стануть абсолютно необхідними (під час `Extract-Min`).

Недоліки

1. Складність реалізації: Купи Фібоначчі значно складніші у реалізації порівняно з бінарними або навіть біноміальними купами. Їхня внутрішня структура з круговими двозв’язними списками для кореневих вузлів та дочірніх вузлів, а також механізми “позначки” та “каскадних вирізань”, вимагають ретельного програмування.

2. Високі константи при невеликій кількості елементів: Хоча амортизована часова складність є дуже привабливою асимптотично, на практиці, для невеликих або середніх розмірів куп, великі приховані константи можуть зробити їх повільнішими, ніж простіші структури даних, такі як бінарні купи. Накладні витрати на керування складною структурою можуть переважати переваги асимптотичної швидкості.

3. Великі накладні витрати на пам’ять: Кожен вузол у купі Фібоначчі зберігає багато посилань (`parent`, `child`, `left_sibling`, `right_sibling`) та додаткові поля (`degree`, `marked`). Це призводить до більшого використання пам’яті порівняно з простішими структурами.

4. Складність налагодження: Через складність внутрішніх механізмів, таких як каскадні вирізання та консолідація, налагодження купи Фібоначчі може бути дуже складним завданням.

5. Рідке використання в загальних випадках: Через вищезгадані недоліки, купи Фібоначчі рідко використовуються в загальних застосунках, якщо немає прямої потреби в їхній специфічній амортизованій продуктивності для `Decrease-Key` або `Union` операцій. У більшості випадків бінарні купи (які простіші та мають менші константи) є достатньо ефективними.

Купи Фібоначчі є потужним інструментом у арсеналі алгоритміста, але їхнє використання виправдане лише тоді, коли асимптотичні переваги для операцій `Decrease-Key` або `Union` є критично важливими для загальної продуктивності алгоритму, особливо для дуже великих обсягів даних. В інших випадках, простіші купи будуть більш доцільними.

Контрольні Запитання

1. Яке основне призначення Купи Фібоначчі?
2. Назвіть ключову відмінність Купи Фібоначчі від звичайної бінарної купи.
3. Опишіть, що таке “амортизована часова складність” і чому вона важлива для Купи Фібоначчі.
4. Які дві основні операції Купи Фібоначчі мають амортизовану часову складність $O(1)$?
5. Які операції Купи Фібоначчі можуть займати $O(\log n)$ амортизованого часу?
6. Поясніть роль `marked` прапорця у вузлі Купи Фібоначчі.
7. Чому операція `Extract-Min` є найскладнішою операцією в Купі Фібоначчі?
8. Яка функція відповідає за зменшення кількості кореневих дерев у Купі Фібоначчі?
9. Наведіть приклад алгоритму, де використання Купи Фібоначчі може значно покращити продуктивність.
10. Які основні недоліки Купи Фібоначчі порівняно з простішими пріоритетними чергами?

Методичні вказівки до лабораторних робіт

1. Назва

Лабораторна робота №X: “Реалізація та Аналіз Купи Фібоначчі”.

2. Мета

Метою даної лабораторної роботи є закріплення теоретичних знань про структуру та операції Купи Фібоначчі, а також набуття практичних навичок

у її реалізації та тестуванні. Робота спрямована на глибоке розуміння амортизованої часової складності та практичних аспектів застосування цієї складної структури даних.

3. Завдання

   а) Реалізувати структуру даних “Купа Фібоначчі” з усіма основними операціями:

   - `Make-Fib-Heap()`
   - `Fib-Heap-Insert(H, x)`
   - `Fib-Heap-Minimum(H)`
   - `Fib-Heap-Extract-Min(H)`
   - `Fib-Heap-Union(H1, H2)`
   - `Fib-Heap-Decrease-Key(H, x, k)`
   - `Fib-Heap-Delete(H, x)`

   б) Реалізувати допоміжні функції: `Fib-Heap-Link(H, y, x)`, `Cut(H, x, y)`, `Cascading-Cut(H, y)`, `Consolidate(H)`.

   в) Розробити тестовий набір для перевірки коректності реалізації кожної операції.

   г) Продемонструвати роботу Купи Фібоначчі на прикладі однієї з її областей застосування (наприклад, спрощений Алгоритм Дейкстри).

4. Порядок Виконання

a) Визначення структури вузла: Створіть клас або структуру `FibHeapNode` (або аналогічну назву) з полями `key`, `degree`, `parent`, `child`, `left_sibling`, `right_sibling`, `marked`.

b) Визначення структури купи: Створіть клас або структуру `FibonacciHeap` (або аналогічну назву) з полями `min_node` (або `H.min`) та `num_nodes` (або `H.n`).

c) Реалізація `Make-Fib-Heap()`: Ініціалізуйте `min_node` як `NULL` і `num_nodes` як `0`.

d) Реалізація `Fib-Heap-Insert(H, x)`: Додайте новий вузол до кореневого списку та оновіть `min_node` за потреби.

e) Реалізація `Fib-Heap-Minimum(H)`: Поверніть `H.min`.

f) Реалізація `Fib-Heap-Link(H, y, x)`: Допоміжна функція для приєднання дерева `y` до `x`.

g) Реалізація `Consolidate(H)`: Ця функція є найскладнішою. Вона використовує допоміжний масив `A` для об'єднання кореневих дерев з однаковим ступенем.

h) Реалізація `Fib-Heap-Extract-Min(H)`: Реалізуйте кроки з переміщенням дочірніх вузлів, видаленням мінімуму та викликом `Consolidate`.

i) Реалізація `Cut(H, x, y)`: Допоміжна функція для від'єднання `x` від `y` та додавання `x` до кореневого списку.

j) Реалізація `Cascading-Cut(H, y)`: Допоміжна функція для каскадних вирізань.

k) Реалізація `Fib-Heap-Decrease-Key(H, x, k)`: Реалізуйте логіку зменшення ключа та виклики `Cut`/`Cascading-Cut`.

l) Реалізація `Fib-Heap-Delete(H, x)`: Використайте `Decrease-Key` та `Extract-Min`.

5. Тестування:

    - Створіть декілька тестових випадків для кожної операції, що охоплюють різні сценарії (порожня купа, купа з одним елементом, купа з багатьма елементами, порушення властивості купи тощо).
    - Використайте функцію обходу дерева (наприклад, прямий обхід) для візуалізації структури купи після ключових операцій (наприклад, після `Extract-Min` або `Decrease-Key`).

6. Демонстрація застосування: Реалізуйте спрощений приклад Алгоритму Дейкстри або Прима, де Купа Фібоначчі використовується як пріоритетна черга. Порівняйте (можна на словах) очікувану продуктивність з використанням бінарної купи.

7. Результат

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де реалізована структура даних “Купа Фібоначчі” з усіма описаними операціями та допоміжними функціями, а також приклади тестування кожної операції. Код повинен бути чистим, добре коментованим та відповідати принципам об’єктно-орієнтованого програмування.
2. Продемонструвати роботу додатку: показати виконання ключових операцій (`Insert`, `Extract-Min`, `Decrease-Key`, `Union`, `Delete`) на конкретних прикладах, а також роботу алгоритму Дейкстри/Прима з використанням реалізованої купи Фібоначчі. Візуалізація структури купи (наприклад, текстова схема або графічний вивід) після критичних операцій буде перевагою.
3. Бути готовим відповісти на питання:
    - Поясніть, як працює механізм `marked` прапорця та `Cascading-Cut`.
    - Яке призначення допоміжного масиву `A` у функції `Consolidate`?
    - Чому `Decrease-Key` в Купі Фібоначчі є амортизовано $O(1)$?
    - Які є компроміси між складністю реалізації та продуктивністю Купи Фібоначчі?
    - Опишіть, як Купа Фібоначчі впливає на часову складність алгоритму Дейкстри.

Тести

1. Яка операція в Купі Фібоначчі завжди виконується за $O(1)$ часу (найгірший випадок і амортизовано)?

    b) `Extract-Min`

    c) `Decrease-Key`

    d) `Insert`

    e) `Delete`

2. Яка операція Купи Фібоначчі відповідає за зменшення кількості кореневих дерев?

    a) `Fib-Heap-Insert`

    b) `Fib-Heap-Union`

    c) `Consolidate`

    d) `Cut`

3. Яке поле вузла Купи Фібоначчі вказує на кількість його прямих дочірніх вузлів?

    a) `parent`

    b) `child`

    c) `degree`

    d) `marked`

4. Для якого алгоритму на графах Купа Фібоначчі може покращити продуктивність завдяки ефективній операції `Decrease-Key`?

    a) Алгоритм Крускала

    b) Алгоритм Дейкстри

    c) Обхід у ширину (BFS)

    d) Топологічне сортування

5. Що відбувається з вузлом, коли його `marked` прапорець встановлюється на `TRUE`?

    a) Він стає кореневим вузлом.

    b) Він позначається як такий, що вже був вирізаний зі свого батька раніше.

    c) Він видаляється з купи.

    d) Його ключ збільшується.

    Відповіді: 1. – d), 2. – c), 3. – c), 4. – c), 5.- b)

Тести підвищеної складності

1. Розгляньте послідовність операцій: n операцій **`Insert`**, а потім одна операція **`Extract-Min`**. Яка амортизована часова складність цієї послідовності для Купи Фібоначчі?

    a) $O(n)$

    b) $O(n\log n)$

    c) $O(n + \log n)$

    d) $O(\log n)$

2. Чому операція **`Fib-Heap-Delete(H, x)`** використовує **`Fib-Heap-Decrease-Key`** перед **`Fib-Heap-Extract-Min`**?

    a) Щоб гарантувати, що **`x`** стане кореневим вузлом.

    b) Щоб **`x`** став мінімальним елементом купи.

    c) Щоб уникнути каскадних вирізань.

    d) Для оптимізації використання пам’яті.

3. Яка властивість зберігається вузлами, що знаходяться у кореневому списку Купи Фібоначчі, після консолідації?

    a) Усі кореневі вузли мають однаковий ступінь.

    b) Кореневі вузли упорядковані за ступенем.

    c) Жоден вузол у кореневому списку не має такого ж ступеня, як інший вузол у кореневому списку.

    d) Усі кореневі вузли є позначеними.

4. Операція **`Cut(H, x, y)`** знімає позначку (**`marked = FALSE`**) з вузла **`x`** після того, як він стає кореневим. Для чого це робиться?

    a) Щоб вказати, що вузол **`x`** більше не має батька.

    b) Щоб скинути його стан для подальших потенційних вирізань.

    c) Це не впливає на подальші операції.

    d) Для зменшення його ступеня до нуля.

5. Припустимо, у вас є купа Фібоначчі, що складається з єдиного дерева, коренем якого є R, а R має k дочірніх вузлів. Яка амортизована часова складність операції `Extract-Min` для цієї купи?

    a) $O(1)$

    b) $O(\log k)$

    c) $O(k)$

    d) $O(\log n)$

6. Яка з наступних структур даних має гіршу амортизовану часову складність для операції `Decrease-Key` порівняно з Купою Фібоначчі?

    a) Біноміальна купа

    b) Бінарна купа

    c) Обидві a) і b)

    d) Жодна з них

7. Яке математичне співвідношення використовується для визначення максимального можливого ступеня кореневого дерева в Купі Фібоначчі з n вузлами?

    a) $O(\log n)$

    b) $O(\log_2 n)$

    c) $O(\log_\phi n)$

    d) $O(\sqrt{n})$

8. Який сценарій може призвести до “каскаду вирізань” у Купі Фібоначчі?

    a) Вставка нового елемента.

    b) Об’єднання двох куп.

    c) Зменшення ключа вузла, який вже був позначений, і його батько не є кореневим.

d) Вилучення мінімального елемента, коли купа складається з одного кореневого дерева.

Відповіді:

1. – a), Кожна з n операцій **`Insert`** має амортизовану складність $O(1)$, що в сумі дає $O(n)$. Операція **`Extract-Min`** має амортизовану складність $O(\log n)$. Отже, загальна амортизована складність послідовності буде $O(n + \log n)$, що спрощується до $O(n)$ для достатньо великих n.

2. – b), Зменшуючи ключ **`x`** до негативної нескінченності, ми гарантуємо, що **`x`** стане найменшим елементом у купі. Після цього **`Extract-Min`** ефективно видалить його.

3. – c), Метою консолідації є об'єднання всіх кореневих дерев з однаковими ступенями, щоб після її завершення в кореневому списку залишалося не більше одного дерева кожного можливого ступеня.

4. - b), Коли вузол **`x`** вирізається і стає кореневим, він більше не має батька, тому його **`marked`** стан стає нерелевантним у цьому контексті. Його скидають, щоб він міг бути позначений знову, якщо він стане дочірнім вузлом і втратить дочірній елемент.

5. – c), Під час **`Extract-Min`**, дочірні вузли мінімального елемента переміщуються до кореневого списку. Якщо мінімальний елемент має k дочірніх вузлів, ця операція додавання цих k вузлів до кореневого списку займе $O(k)$ часу. Подальша консолідація матиме амортизовану складність, але переміщення дочірніх вузлів — це $O(k)$ в найгіршому випадку.

6. – a), b) І біноміальна купа, і бінарна купа мають амортизовану часову складність $O(\log n)$ для операції **`Decrease-Key`**, тоді як Купа Фібоначчі має $O(1)$.

7. – c), Максимальний ступінь кореневого дерева в Купі Фібоначчі з n вузлами пов'язаний з логарифмом за основою золотого перетину $\phi \approx 1.618$.

8. – c), Якщо вузол **`x`** позначений і його ключ зменшується так, що він порушує властивість купи, **`x`** вирізається. Потім викликається

`Cascading-Cut` для його батька. Якщо батько `y` також був позначений, він також буде вирізаний, і процес продовжиться вгору по дереву, потенційно викликаючи каскад вирізань аж до кореня або до непозначеного вузла.

Завдання для самостійної роботи

1. Дослідження часової складності: Детально вивчіть, чому операції `Insert`, `Minimum`, `Union` та `Decrease-Key` у Купі Фібоначчі мають амортизовану часову складність $O(1)$, а `Extract-Min` та `Delete` — $O(\log n)$. Поясніть концепцію амортизованого аналізу.

2. Порівняльний аналіз: Проведіть порівняльний аналіз Купи Фібоначчі з бінарною купою та біноміальною купою за такими критеріями: часова складність для всіх операцій (найгірший випадок та амортизовано), складність реалізації, використання пам’яті. Створіть таблицю з порівняльними характеристиками.

3. Вплив констант: Дослідіть вплив прихованих констант на практичну продуктивність Купи Фібоначчі. Знайдіть джерела або статті, які обговорюють, коли Купа Фібоначчі починає перевершувати бінарні купи на практиці (наприклад, для яких розмірів графа в алгоритмі Дейкстри).

4. Реалізація Алгоритму Прима: Реалізуйте алгоритм Прима для пошуку мінімального остовного дерева, використовуючи вашу реалізацію Купи Фібоначчі як пріоритетну чергу. Протестуйте його на кількох тестових графах.

5. Реалізація Алгоритму Дейкстри: Реалізуйте алгоритм Дейкстри для пошуку найкоротшого шляху, використовуючи вашу реалізацію Купи Фібоначчі як пріоритетну чергу. Протестуйте його на кількох тестових графах.

6. Модифікація `Delete`: Розгляньте можливість реалізації операції `Delete(H, x)` без попереднього виклику `Decrease-Key`. Опишіть, як це вплине на складність та логіку алгоритму. Чи буде це ефективніше?

7. Генерація тестових даних: Розробіть програму для автоматичної генерації випадкових графів для тестування алгоритмів Дейкстри та

Прима з різною кількістю вершин та ребер. Це допоможе оцінити продуктивність Купи Фібоначчі на різних вхідних даних.

8. Візуалізація купи: Розробіть інструмент (наприклад, консольний вивід або проста графічна бібліотека), який дозволить візуалізувати структуру Купи Фібоначчі після кожної значної операції (`Insert`, `Extract-Min`, `Consolidate`). Це допоможе краще зрозуміти її динамічну поведінку.
9. Оптимізація пам'яті: Запропонуйте та обговоріть можливі підходи до оптимізації використання пам'яті у Купі Фібоначчі. Чи можна зменшити кількість посилань або прапорців без значної втрати продуктивності?
10. Застосування у реальному світі: Дослідіть та опишіть, окрім алгоритмів на графах, інші потенційні або вже існуючі застосування Куп Фібоначчі в реальних системах або складних обчислювальних задачах.

Подвійна купа

Розглянемо купу подвійна (Pairing Heap) – просту у реалізації та практично ефективну структуру даних, яка належить до класу узагальнених куп. Вона поєднує в собі переваги простоти бінарних куп і хорошу амортизовану продуктивність, що робить її привабливою альтернативою для багатьох застосувань, де важливі операції об'єднання та зменшення ключа.

Подвійна купа — це міні-купа на основі дерев, яка підтримує властивість міні-купи: ключ кожного вузла менший або дорівнює ключам усіх його дочірніх вузлів.

На відміну від куп Фібоначчі, які є сукупністю дерев, купа подвійна, як правило, складається з одного дерева. Її структура є більш гнучкою, ніж біноміальні купи, але менш складною, ніж купи Фібоначчі.

Кожен вузол у купі подвійна містить наступну інформацію:

- **`ключ`** (key): Значення вузла, за яким він упорядковується в купі (наприклад, пріоритет).
- **`дочірній_вузол`** (child): Посилання на перший дочірній вузол.

– **лівий_брат** (left_sibling) та **правий_брат** (right_sibling): Посилання на лівого та правого братів у двозв'язному списку, який утворюють дочірні вузли одного батька. Цей список не круговий.

– **батько** (parent): Посилання на батьківський вузол. Для кореневого вузла це посилання буде **NULL**.

Ключова ідея подвійної купи полягає в тому, що всі дочірні вузли одного батька організовані у двозв'язний список. Це дозволяє ефективно додавати та видаляти дочірні вузли.

Операції купи

Подвійні купи підтримують стандартні операції пріоритетної черги, при цьому багато з них мають хорошу амортизовану часову складність:

1. **Make-Pairing-Heap()**: Створення порожньої купи
2. **Pairing-Heap-Insert(H, x)**: Вставка нового вузла
3. **Pairing-Heap-Minimum(H)**: Пошук мінімального елемента
4. **Pairing-Heap-Extract-Min(H)**: Вилучення мінімального елемента
5. **Pairing-Heap-Meld(H1, H2)**: Об'єднання двох куп
6. **Pairing-Heap-Decrease-Key(H, x, k)**: Зменшення ключа вузла
7. **Pairing-Heap-Delete(H, x)**: Видалення вузла

Давайте детально розглянемо кожну з цих операцій.

Створення порожньої подвійної купи Make-Pairing-Heap()

Ця операція створює та повертає нову, порожню купу подвійна. Вона просто ініціалізує посилання на корінь купи.

Опис: Подвійна купа **H** може бути представлена як посилання на її кореневий вузол. Якщо купа порожня, це посилання буде **NULL**.

Візуалізація роботи: Уявіть порожню область, де ще немає жодного вузла. Це візуалізація порожньої подвійної купи .

Подвійна купа H
H.root = NULL

Алгоритм:

```
Make-Pairing-Heap()
  H.root = NULL  // Ініціалізуємо посилання на кореневий вузол як NUL
L
  Повернути H     // Повертаємо порожню купу
```

Псевдокод:

```
FUNCTION Make-Pairing-Heap():
  H = NEW PairingHeapObject
  H.root = NIL
  RETURN H
```

Вставка нового вузла `Pairing-Heap-Insert(H, x)`

Ця операція додає новий вузол `x` до подвійної купи `H`. Вставлений вузол стає новим коренем, якщо його ключ менший за поточний корінь. Це досягається за допомогою операції `Meld`.

Опис: Для вставки нового вузла `x` необхідно виконати наступні кроки:

1. Ініціалізація вузла: Створити новий вузол `x` з заданим ключем. Всі його посилання на дочірні елементи, братів та батька ініціалізуються як `NULL`.
    - `x.child = NULL`
    - `x.left_sibling = NULL`
    - `x.right_sibling = NULL`
    - `x.parent = NULL`
2. Об'єднання: Якщо купа `H` не порожня, то новий вузол `x` об'єднується з поточним коренем купи `H.root` за допомогою допоміжної функції `Meld`. Результатом `Meld` є новий корінь купи. Якщо купа `H` порожня, `x` просто стає коренем.

Припустимо, у нас є купа `H` з коренем `A` (ключ 10). Ми хочемо вставити новий вузол `B` (ключ 5).

До вставки:

```
  H.root -> (A, 10)
             /  \
            ...
```

Після вставки B (використовуючи Meld):

Оскільки ключ B менший за ключ A, B стає новим коренем, а A стає дочірнім вузлом B.

```
  H.root -> (B, 5)
             |
             v
            (A, 10)
             /  \
            ...
```

Алгоритм:

```
Pairing-Heap-Insert(H, x)
  x.child = NULL
  x.left_sibling = NULL
  x.right_sibling = NULL
  x.parent = NULL

  Якщо H.root == NULL:
    H.root = x
  Інакше:
    H.root = Meld(H.root, x) // Об'єднуємо x з поточним коренем H
```

Псевдокод такий:

```
FUNCTION Pairing-Heap-Insert(H, x):
  // Ініціалізація вузла x
  x.child = NIL
  x.left_sibling = NIL
  x.right_sibling = NIL
  x.parent = NIL

  IF H.root IS NIL:
    H.root = x
  ELSE:
    H.root = Meld(H.root, x)
```

Пошук мінімального елемента `Pairing-Heap-Minimum(H)`

Ця операція повертає вузол з найменшим ключем у купі подвійна **H**. Завдяки властивості купи, мінімальний елемент завжди є коренем дерева.

Опис: Подвійна купа зберігає посилання на кореневий вузол у полі **`H.root`**. Оскільки корінь завжди має найменший ключ серед усіх вузлів у купі, достатньо просто повернути значення цього поля. Ця операція займає константний час $O(1)$.

Припустимо, у нас є купа `H` з коренем `B`.

```
H.root -> (B, 5)  (мінімальний)
           |
           v
          (A, 10)
           / \
          ...
```

Алгоритм:

```
Pairing-Heap-Minimum(H)
  Повернути H.root
```

Псевдокод такий:

```
FUNCTION Pairing-Heap-Minimum(H):
  RETURN H.root
```

Видалення мінімального елемента `Pairing-Heap-Extract-Min(H)`

> Ця операція видаляє вузол з найменшим ключем (корінь) з подвійної купи `H` і повертає його. Це найскладніша операція, оскільки вона вимагає перебудови купи після видалення кореня.

Опис: Видалення мінімального елемента включає такі кроки:

1. Збереження кореня: Зберегти посилання на поточний кореневий вузол `z = H.root`.
2. Перевірка порожньої купи: Якщо `z` є `NULL`, купа порожня, повертаємо `NULL`.
3. Оновлення кореня:
    - Якщо `z.child` є `NULL` (тобто `z` був єдиним вузлом у купі), то `H.root` стає `NULL`.
    - Інакше, дочірні вузли `z` необхідно об’єднати, щоб сформувати новий корінь купи. Для цього використовується допоміжна функція `Two-Pass-Meld`. Ця функція обробляє список дочірніх вузлів у два проходи:

- Перший прохід (left-to-right): Попарно об’єднуються сусідні дочірні вузли. Результати об’єднання зберігаються у тимчасовому списку.
- Другий прохід (right-to-left): Результати першого проходу об’єднуються між собою, рухаючись справа наліво, формуючи єдине дерево, яке стане новим коренем купи.

4. Зняття батьківських посилань: Встановити `parent` для всіх дочірніх вузлів `z`, які стають новим коренем, на `NULL`.

Об'єднання списку коренів (внутрішня допоміжна функція) `Two-Pass-Meld(list_of_nodes)`:

Ця функція об'єднує список коренів (які є дочірніми вузлами видаленого мінімального елемента) в одне єдине дерево, яке стане новим коренем купи.

Опис `Two-Pass-Meld`:

1. Базовий випадок: Якщо список порожній, повертаємо `NULL`. Якщо список містить лише один вузол, повертаємо цей вузол.
2. Перший прохід (зліва направо):
    - Проходимо по списку вузлів парами.
    - Для кожної пари вузлів (`node1`, `node2`) викликаємо `Meld(node1, node2)`. Результатом є одне дерево.
    - Зберігаємо ці об’єднані дерева у новому тимчасовому списку. Якщо кількість вузлів у вихідному списку була непарною, останній вузол, який не мав пари, додається до тимчасового списку як єдиний елемент.
3. Другий прохід (справа наліво):
    - Проходимо по тимчасовому списку об’єднаних дерев у зворотному порядку.
    - Об’єднуємо дерева одне за одним за допомогою `Meld`. Перше об’єднання відбувається між двома останніми деревами, потім результат об’єднується з передостаннім і так далі.
    - Останній результат об’єднання буде новим коренем купи.

Об'єднання двох дерев (внутрішня допоміжна функція) `Meld(H1, H2)`

Ця функція об'єднує два дерева `H1` і `H2`, які самі є коренями міні-куп, в одне дерево, зберігаючи властивість міні-купи.

Опис `Meld(H1, H2)`:

1. Порівняння коренів: Визначаємо, який з коренів (`H1.key` або `H2.key`) менший. Вузол з меншим ключем стає батьком.
2. Приєднання меншого до більшого:
    - Якщо `H1.key <= H2.key`: `H2` стає дочірнім вузлом `H1`. `H1` залишається коренем.
    - Якщо `H2.key < H1.key`: `H1` стає дочірнім вузлом `H2`. `H2` залишається коренем.
3. Оновлення посилань:
    - Новий дочірній вузол (`H2` або `H1` відповідно) додається до початку двозв'язного списку дочірніх вузлів нового батька.
    - Встановлюється посилання `parent` для нового дочірнього вузла.

Розглянемо роботу `Extract-Min`:

Нехай до роботи `Extract-Min`:

```
H.root -> (A, 5)
           |
           v
          (B, 10) -- (C, 15) -- (D, 7)  (дочірні вузли A)
           / \         / \        / \
          (E,20)   (F,8)(G,12) (H,9)(I,11)
```

Крок 1: Зберегти A, потім обробити дочірні B, C, D за допомогою `Two-Pass-Meld` (внутрішня допоміжна функція).

Перший прохід `Two-Pass-Meld` (зліва направо):

– `Meld(B, C)` -> Результат `BC_merged` (корінь B, C - дочірній)

– `D` залишається сам

Проміжний список: `BC_merged`, `D`

Другий прохід `Two-Pass-Meld` (справа наліво):

– `Meld(D, BC_merged)` -> Результат `New_Root` (корінь D, `BC_merged` - дочірній)

Після `Extract-Min`:

```
H.root -> (D, 7)  (новий корінь)
           |
           v
          (B, 10)
           / \
          (C, 15)  (E, 20)
           / \
          (F, 8) (G, 12)
           / \
          (H, 9) (I, 11)
```

(точна структура після `Meld` залежить від значень ключів та порядку приєднання. Наприклад, якщо B < C, то B буде батьком, а C його дочірнім. Якщо D < B, то D буде батьком, а B його дочірнім, і так далі.)

Повернемось до алгоритму `Pairing-Heap-Extract-Min(H)`:

```
Pairing-Heap-Extract-Min(H)
  z = H.root
  Якщо z == NULL:
    Повернути NULL

  Якщо z.child == NULL: // Корінь був єдиним вузлом
    H.root = NULL
  Інакше:
    // Зняти батьківські посилання з дочірніх вузлів
    current_child = z.child
    Повторювати:
      current_child.parent = NULL
      current_child = current_child.right_sibling
    Поки current_child != NULL

    H.root = Two-Pass-Meld(z.child) // Злиття дочірніх вузлів

  // Очистити посилання з видаленого вузла для коректного повернення
  z.child = NULL
  z.left_sibling = NULL
  z.right_sibling = NULL
  z.parent = NULL
```

```
  Повернути z
```

Алгоритм `Two-Pass-Meld(list_of_nodes)`: (Приймає посилання на перший вузол списку)

```
Two-Pass-Meld(first_node_in_list)
  Якщо first_node_in_list == NULL:
    Повернути NULL
  Якщо first_node_in_list.right_sibling == NULL: // Лише один вузол у списку
    Повернути first_node_in_list

  // Перший прохід (зліва направо)
  paired_list = []
  current = first_node_in_list
  Поки current != NULL:
    next_node = current.right_sibling
    current.right_sibling = NULL // Від'єднуємо від списку
    Якщо next_node != NULL:
      temp = next_node.right_sibling
      next_node.right_sibling = NULL // Від'єднуємо від списку
      paired_list.Додати(Meld(current, next_node))
      current = temp
    Інакше:
      paired_list.Додати(current) // Непарний вузол
      current = NULL

  // Другий прохід (справа наліво)
  Якщо paired_list.Розмір == 1:
    Повернути paired_list[0]

  result = paired_list[paired_list.Розмір - 1]
  для i від paired_list.Розмір - 2 до 0 (з кроком -1):
    result = Meld(result, paired_list[i])
  Повернути result
```

Алгоритм `Meld(H1, H2)`:

```
Meld(H1, H2)
  Якщо H1 == NULL:
    Повернути H2
  Якщо H2 == NULL:
    Повернути H1

  Якщо H1.key <= H2.key:
```

```
    // H2 стає дочірнім вузлом H1
    H2.parent = H1
    H2.right_sibling = H1.child
    Якщо H1.child != NULL:
      H1.child.left_sibling = H2
    H1.child = H2
    Повернути H1
  Інакше:
    // H1 стає дочірнім вузлом H2
    H1.parent = H2
    H1.right_sibling = H2.child
    Якщо H2.child != NULL:
      H2.child.left_sibling = H1
    H2.child = H1
    Повернути H2
```

Псевдокод `Pairing-Heap-Extract-Min(H)` такий:

```
FUNCTION Pairing-Heap-Extract-Min(H):
  z = H.root
  IF z IS NIL:
    RETURN NIL

  IF z.child IS NIL:
    H.root = NIL
  ELSE:
    // Зняти батьківські посилання з дочірніх вузлів
    current_child = z.child
    WHILE current_child IS NOT NIL:
      current_child.parent = NIL
      current_child = current_child.right_sibling

    H.root = Two-Pass-Meld(z.child) // Злиття дочірніх вузлів

  // Очистити посилання з видаленого вузла
  z.child = NIL
  z.left_sibling = NIL
  z.right_sibling = NIL
  z.parent = NIL

  RETURN z
```

Псевдокод `Two-Pass-Meld(first_node_in_list)` такий:

```
FUNCTION Two-Pass-Meld(first_node_in_list):
  IF first_node_in_list IS NIL:
```

```
    RETURN NIL
  IF first_node_in_list.right_sibling IS NIL: // Лише один вузол у сп
иску
    RETURN first_node_in_list

  // Зібрати вузли в список для зручності
  node_list = LIST()
  current = first_node_in_list
  WHILE current IS NOT NIL:
    node_list.ADD(current)
    current = current.right_sibling

  // Перший прохід (зліва направо)
  paired_results = LIST()
  i = 0
  WHILE i < node_list.SIZE:
    node1 = node_list[i]
    node1.right_sibling = NIL // Від'єднуємо від оригінального списку
    node1.left_sibling = NIL // Від'єднуємо
    IF i + 1 < node_list.SIZE:
      node2 = node_list[i + 1]
      node2.right_sibling = NIL
      node2.left_sibling = NIL
      paired_results.ADD(Meld(node1, node2))
      i = i + 2
    ELSE:
      paired_results.ADD(node1) // Непарний вузол
      i = i + 1

  // Другий прохід (справа наліво)
  IF paired_results.SIZE == 1:
    RETURN paired_results[0]

  result = paired_results[paired_results.SIZE - 1]
  FOR i FROM paired_results.SIZE - 2 DOWNTO 0:
    result = Meld(paired_results[i], result) // Важливо: Meld(менший,
більший) або (H1, H2) в залежності від реалізації
  RETURN result
```

Псевдокод `Meld(H1, H2)`:

```
FUNCTION Meld(H1, H2):
  IF H1 IS NIL:
    RETURN H2
  IF H2 IS NIL:
    RETURN H1
```

```
IF H1.key <= H2.key:
  // H2 стає дочірнім вузлом H1
  H2.parent = H1
  // Вставляємо H2 на початок списку дочірніх вузлів H1
  H2.right_sibling = H1.child
  IF H1.child IS NOT NIL:
    H1.child.left_sibling = H2
  H1.child = H2
  RETURN H1
ELSE:
  // H1 стає дочірнім вузлом H2
  H1.parent = H2
  // Вставляємо H1 на початок списку дочірніх вузлів H2
  H1.right_sibling = H2.child
  IF H2.child IS NOT NIL:
    H2.child.left_sibling = H1
  H2.child = H1
  RETURN H2
```

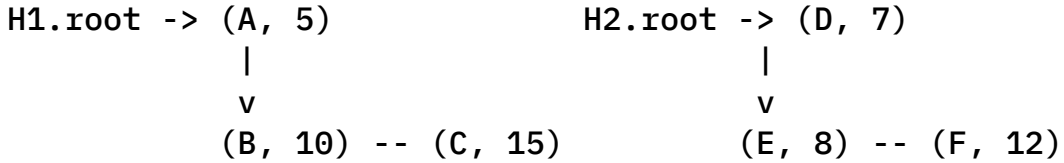
Об’єднання двох подвійних куп `Pairing-Heap-Meld(H1, H2)`

Ця операція об’єднує дві подвійні купи **`H1`** та **`H2`** в одну нову купу. Вона є дуже ефективною, оскільки полягає лише в об’єднанні їхніх коренів.

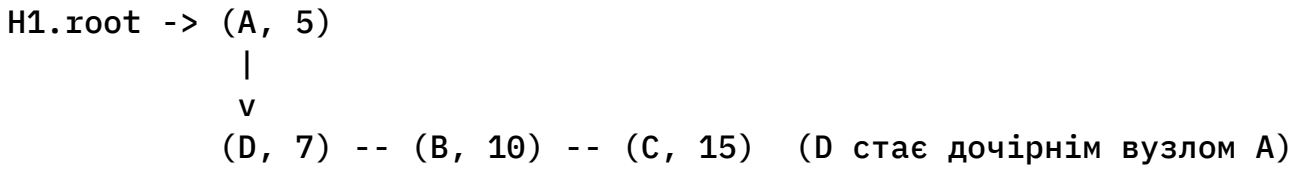
Опис: Для об’єднання двох куп **`H1`** та **`H2`**:

1. Виклик **`Meld`**: Просто викликається допоміжна функція **`Meld`** для коренів **`H1.root`** та **`H2.root`**.
2. Оновлення **`H1`**: Результат **`Meld`** стає новим коренем **`H1`**. **`H2`** стає порожньою купою .

Нехай до об’єднання купи такі:



Після об’єднання (припустимо, A < D):



```
          v
        (E, 8) -- (F, 12)
```

Зверніть увагу, що D зі своїми дочірніми елементами тепер знаходиться у списку дочірніх елементів A.

Алгоритм:

```
Pairing-Heap-Meld(H1, H2)
  H1.root = Meld(H1.root, H2.root)
  H2.root = NULL // H2 стає порожньою
  Повернути H1    // Можна повернути H1, або просто змінити H1 in-pla
ce
```

Псевдокод:

```
FUNCTION Pairing-Heap-Meld(H1, H2):
  H1.root = Meld(H1.root, H2.root)
  H2.root = NIL
  RETURN H1 // Або просто оновлюємо H1, залежить від інтерфейсу
```

Зменшення ключа вузла `Pairing-Heap-Decrease-Key(H, x, k)`

Ця операція зменшує ключ вузла `x` до нового значення `k`. Важливо, що `k` повинно бути меншим за поточний ключ `x.key`. Якщо після зменшення ключа властивість міні-купи порушується (тобто `x.key < x.parent.key`), то вузол `x` “вирізається” зі свого батьківського дерева і об’єднується з коренем купи.

Опис:

1. Перевірка вхідних даних: Переконатися, що новий ключ `k` дійсно менший за поточний ключ `x.key`. Якщо ні, це є помилкою.
2. Оновлення ключа: Встановити `x.key = k`.
3. Перевірка порушення властивості купи: Якщо `x` не є коренем купи (`x.parent` не `NULL`) і `x.key < x.parent.key`:
    - Видалення з батьківського списку: Видалити `x` з двозв’язного списку дочірніх вузлів `x.parent`.
    - Оновлення посилань: `x.parent` стає `NULL`, а `x.left_sibling` та `x.right_sibling` також стають `NULL` (він стає окремим деревом).

- Об’єднання з коренем: Об’єднати `x` (який тепер є окремим деревом) з поточним коренем купи `H.root` за допомогою `Meld`. Результат `Meld` стає новим `H.root`.

Розглянемо роботу `Decrease-Key` .

Нехай до `Decrease-Key(H, G, 6)` (G має ключ 12, батько F):

```
H.root -> (A, 5)
           |
           v
          (F, 8) -- (B, 10)
           |
           v
          (G, 12) -- (C, 15)
```

Після `Decrease-Key(H, G, 6)`:

`G.key` стає `6`. Тепер `G.key (6) < F.key (8)`. Властивість купи порушена.

1. `G` видаляється з дочірніх вузлів `F`.
2. `G.parent = NULL`, `G.left_sibling = NULL`, `G.right_sibling = NULL`. `G` стає окремим деревом.
3. `Meld(H.root, G)`: `H.root` (A) і `G` об’єднуються. Оскільки `G.key (6)` > `A.key (5)`, `G` стає дочірнім вузлом `A`.

Після `Decrease-Key`:

```
H.root -> (A, 5)
           |
           v
          (G, 6) -- (F, 8) -- (B, 10)  (G вставлено на початок спис
ку дочірніх вузлів A)
                     |
                     v
                    (C, 15)
```

Алгоритм `Pairing-Heap-Decrease-Key(H, x, k)`:

```
Pairing-Heap-Decrease-Key(H, x, k)
  Якщо k > x.key:
    Помилка "Новий ключ більший за поточний"

  x.key = k

  Якщо x == H.root: // x вже є коренем, нічого не треба робити, крім
```

```
оновлення ключа
    Повернути

  y = x.parent

  // Видалити x з дочірнього списку його батька y
  Якщо x.left_sibling != NULL:
    x.left_sibling.right_sibling = x.right_sibling
  Інакше: // x був першим дочірнім вузлом y
    y.child = x.right_sibling

  Якщо x.right_sibling != NULL:
    x.right_sibling.left_sibling = x.left_sibling

  // x тепер є окремим деревом
  x.left_sibling = NULL
  x.right_sibling = NULL
  x.parent = NULL

  // Об'єднати x з коренем купи
  H.root = Meld(H.root, x)
```

Псевдокод `Pairing-Heap-Decrease-Key(H, x, k)` такий:

```
FUNCTION Pairing-Heap-Decrease-Key(H, x, k):
  IF k > x.key:
    ERROR "New key is greater than current key."

  x.key = k

  IF x IS H.root: // x вже корінь, тільки ключ оновили
    RETURN

  y = x.parent

  // Видалення x з дочірнього списку y
  IF x.left_sibling IS NOT NIL:
    x.left_sibling.right_sibling = x.right_sibling
  ELSE: // x був першим дочірнім вузлом y
    y.child = x.right_sibling

  IF x.right_sibling IS NOT NIL:
    x.right_sibling.left_sibling = x.left_sibling

  // x тепер є окремим деревом
```

```
  x.left_sibling = NIL
  x.right_sibling = NIL
  x.parent = NIL

  // Об'єднати x з коренем купи
  H.root = Meld(H.root, x)
```

Видалення вузла `Pairing-Heap-Delete(H, x)`

Ця операція видаляє довільний вузол **x** з подвійної купи **H**. Вона реалізується шляхом зменшення ключа **x** до негативної нескінченності (або дуже малого значення), що гарантує, що **x** стане мінімальним елементом, а потім викликом **Extract-Min**.

Опис:

1. Зменшення ключа: Викликати **Pairing-Heap-Decrease-Key(H, x, -infinity)** (або дуже мале число, наприклад, **MIN_INT**). Це перемістить **x** до кореня купи (якщо він не був коренем) і зробить його новим мінімальним елементом.
2. Вилучення мінімуму: Викликати **Pairing-Heap-Extract-Min(H)**. Це видалить **x** (який тепер є мінімальним елементом) з купи.

Розглянемо роботу **Delete**:

Нехай до **Delete(H, G)** (G має ключ 12):

```
  H.root -> (A, 5)
             |
             v
            (F, 8) -- (B, 10)
             |
             v
            (G, 12) -- (C, 15)
```

Крок 1: `Decrease-Key(H, G, -infinity)`

G стає кореневим вузлом з дуже малим ключем, і **H.root** вказує на **G**.

```
  H.root -> (G, -infinity)
             |
             v
            (A, 5)  (A стає дочірнім вузлом G, або G дочірнім A, а по тім H.root = A)
             |
             v
```

```
            (F, 8) -- (B, 10)
             |
             v
            (C, 15)
```

Після `Decrease-Key`, `G` буде об'єднано з поточним `H.root` (яким був `A`). Оскільки `G.key` тепер $-\infty$, `G` стане новим `H.root`, а `A` стане його дочірнім вузлом.)

Крок 2: `Extract-Min(H)`

`G` (який тепер є мінімальним) видаляється. Дочірні вузли `G` (серед яких `A` з усією його гілкою) обробляються за допомогою `Two-Pass-Meld`, щоб сформувати новий корінь купи.

Після `Delete`:

```
  H.root -> (A, 5)  (якщо A був єдиним дочірнім G)
             |
             v
            (F, 8) -- (B, 10)
             |
             v
            (C, 15)
```

Алгоритм:

```
Pairing-Heap-Delete(H, x)
  Pairing-Heap-Decrease-Key(H, x, -infinity) // Зменшуємо ключ до най
меншого можливого значення
  Pairing-Heap-Extract-Min(H) // Видаляємо теперішній мінімальний еле
мент (який є x)
```

Псевдокод такий:

```
FUNCTION Pairing-Heap-Delete(H, x):
  // Припускаємо, що є константа NEGATIVE_INFINITY
  Pairing-Heap-Decrease-Key(H, x, NEGATIVE_INFINITY)
  Pairing-Heap-Extract-Min(H)
```

Обходи

Як і для купи Фібоначчі, обхід подвійної купи стосується обходу її єдиного дерева. Оскільки дочірні вузли організовані у двозв'язний список, обходи у глибину (DFS) та обхід у ширину (BFS) можуть бути реалізовані.

Структура вузла для обходу

```
Структура Вузла:
  key: int
  parent: Node*
  child: Node* // Посилання на першого дочірнього вузла
  left_sibling: Node*
  right_sibling: Node*
```

3. Обхід у глибину (Depth-First Search - DFS)

1.1. Прямий обхід

Відвідати кореневий вузол, потім рекурсивно відвідати всі його дочірні піддерева (переглядаючи їх у списку дочірніх вузлів).

Візуалізація роботи:

```
      A
     /|\
    B-C-D  (B, C, D - брати, пов'язані списком)

Порядок обходу: A -> B -> C -> D (якщо B перший дочірній)
```

Алгоритм:

```
Pre-order-Traversal(node)
  Якщо node == NULL:
    Повернути
  Відвідати(node) // Наприклад, вивести node.key

  current_child = node.child
  Поки current_child != NULL:
    Pre-order-Traversal(current_child)
    current_child = current_child.right_sibling
```

Псевдокод:

```
PROCEDURE Pre-order-Traversal(node):
  IF node IS NIL:
    RETURN

  VISIT(node) // Наприклад, PRINT node.key

  current_child = node.child
  WHILE current_child IS NOT NIL:
    Pre-order-Traversal(current_child)
    current_child = current_child.right_sibling
```

1.2. Зворотний обхід

Рекурсивно відвідати всі дочірні піддерева, а потім відвідати кореневий вузол.

Візуалізація роботи:

```
    A
   /|\
  B-C-D

Порядок обходу: B -> C -> D -> A
```

Алгоритм:

```
Post-order-Traversal(node)
  Якщо node == NULL:
    Повернути

  current_child = node.child
  Поки current_child != NULL:
    Post-order-Traversal(current_child)
    current_child = current_child.right_sibling

  Відвідати(node) // Наприклад, вивести node.key
```

Псевдокод такий:

```
PROCEDURE Post-order-Traversal(node):
  IF node IS NIL:
    RETURN

  current_child = node.child
  WHILE current_child IS NOT NIL:
    Post-order-Traversal(current_child)
    current_child = current_child.right_sibling

  VISIT(node) // Наприклад, PRINT node.key
```

1.4. Обхід у ширину

Відвідати вузли рівень за рівнем, починаючи від кореня. Використовує чергу для відстеження вузлів, які потрібно відвідати.

Візуалізація роботи:

```
    A
   /|\
  B-C-D
```

```
   / \
  E   F

Порядок обходу: A -> B -> C -> D -> E -> F
```

Алгоритм:

```
BFS-Traversal(root_node)
  Якщо root_node == NULL:
    Повернути

  queue = Нова_Черга()
  queue.Додати(root_node)

  Поки queue не порожня:
    current_node = queue.Видалити()
    Відвідати(current_node) // Наприклад, вивести current_node.key

    current_child = current_node.child
    Поки current_child != NULL:
      queue.Додати(current_child)
      current_child = current_child.right_sibling
```

Псевдокод такий:

```
PROCEDURE BFS-Traversal(root_node):
  IF root_node IS NIL:
    RETURN

  queue = NEW QUEUE()
  queue.ENQUEUE(root_node)

  WHILE NOT queue.IS_EMPTY():
    current_node = queue.DEQUEUE()
    VISIT(current_node) // Наприклад, PRINT current_node.key

    current_child = current_node.child
    WHILE current_child IS NOT NIL:
      queue.ENQUEUE(current_child)
      current_child = current_child.right_sibling
```

Обхід всієї подвійної купи

Оскільки подвійна купа є одним деревом, для обходу всієї купи достатньо викликати відповідну функцію обходу для її кореня:

```
Pairing-Heap-Full-Traversal(H)
  Pre-order-Traversal(H.root) // Або Post-order-Traversal, або BFS-Traversal
```

2. Область застосування подвійних куп

Подвійні купи є чудовим вибором для багатьох алгоритмів і систем, де потрібна ефективна пріоритетна черга. Їхня простота реалізації в поєднанні з хорошою амортизованою продуктивністю робить їх практично конкурентоспроможними з більш складними структурами, такими як купи Фібоначчі.

Основними областями застосування є:

- Алгоритми на графах: Це основна область, де подвійні купи можуть значно покращити продуктивність. Вони ефективні у алгоритмах, що використовують операції “зменшення ключа” та “вилучення мінімуму”.
- Мережеві алгоритми: У маршрутизації, аналізі мереж та інших задачах, де необхідно динамічно оновлювати пріоритети вузлів або ребер.
- Симуляції дискретних подій: У системах, де потрібно керувати чергою подій, що відбуваються в певний час, і де час події може змінюватися.
- Задачах оптимізації: Загальні задачі, де пріоритети елементів можуть динамічно змінюватися.
- Компілятори та операційні системи: У деяких внутрішніх структурах даних для планування завдань або керування пам’яттю.

3. Приклади застосування подвійних куп

Давайте розглянемо конкретні приклади алгоритмів, де подвійні купи демонструють свою ефективність.

3.1. Алгоритм Дейкстри для пошуку найкоротшого шляху

Алгоритм Дейкстри призначений для пошуку найкоротших шляхів від однієї початкової вершини до всіх інших вершин у графі з невід’ємними вагами ребер. Його ефективність сильно залежить від продуктивності пріоритетної черги.

Під час роботи алгоритму, при знаходженні коротшого шляху до вже відкритої вершини, її пріоритет (відстань) у черзі необхідно зменшити. Подвійна купа є дуже ефективною для цієї операції.

- Крок 1: Ініціалізація. Всі вершини графа додаються до подвійної купи, де ключем є поточна відома відстань до вершини (для початкової вершини - 0, для решти - нескінченність).
- Крок 2: Ітерації. Доки подвійна купа не порожня:
  - Вилучення мінімуму: Вершина u з найменшою відстанню вилучається (`Pairing-Heap-Extract-Min`).
  - Оновлення сусідів: Для кожного сусіда v вершини u: якщо новий шлях до v через u коротший, ніж поточний відомий шлях до v, то відстань до v оновлюється, і викликається операція `Pairing-Heap-Decrease-Key` для v у подвійній купі.

Хоча теоретична амортизована складність `Decrease-Key` для подвійної купи є $O(\log n)$ (існує суперечка щодо точної константи, але вона краща, ніж $O(\log n)$ у бінарних купах), на практиці вона часто працює набагато швидше, близькою до $O(1)$, завдяки простій структурі та ефективній операції `Meld`. Це робить алгоритм Дейкстри з подвійною купою дуже конкурентоспроможним, часто досягаючи продуктивності, подібної до Куп Фібоначчі ($O(E + V\log V)$), але зі значно простішою реалізацією.

3.2. Алгоритм Прима для побудови мінімального остовного дерева

Алгоритм Прима будує мінімальне остовне дерево (MST) у зв’язаному, зваженому, неорієнтованому графі. Він також активно використовує пріоритетну чергу.

Аналогічно алгоритму Дейкстри, алгоритм Прима вимагає ефективного оновлення “вартості” вершин (мінімальної ваги ребра, що з’єднує вершину з уже побудованою частиною MST).

- Крок 1: Ініціалізація. Всі вершини додаються до подвійної купи. Ключем є мінімальна вага ребра, що з’єднує вершину з початковою вершиною MST (або нескінченність, якщо такого ребра немає).
- Крок 2: Ітерації. Доки подвійна купа не порожня:
  - Вилучення мінімуму: Вершина u з найменшим ключем вилучається (`Pairing-Heap-Extract-Min`).

- Оновлення сусідів: Для кожного сусіда v вершини u: якщо v ще не в MST і вага ребра (u, v) менша за поточний ключ v, то ключ v оновлюється до ваги (u, v), і викликається операція `Pairing-Heap-Decrease-Key` для v.

Подвійна купа забезпечує практично високу продуктивність для алгоритму Прима, подібну до тієї, що досягається за допомогою куп Фібоначчі.

3.3. Реалізація інших алгоритмів на графах

Подвійні купи також можуть бути використані в інших алгоритмах на графах, які потребують ефективних пріоритетних черг, таких як:

– Алгоритм пошуку k-найкоротших шляхів: Де потрібно підтримувати кілька найкоротших шляхів і постійно оновлювати їх.

– Алгоритми обчислення мережевих потоків: У деяких ітеративних методах для знаходження максимального потоку або мінімального розрізу.

Переваги та Недоліки

Подвійні купи є цікавим компромісом між складністю та продуктивністю.

Переваги

1. Простота реалізації: Порівняно з купами Фібоначчі, подвійної купи значно простіші в реалізації. Їхня структура з двозв'язними списками дочірніх вузлів та двопрохідним об'єднанням є легшою для розуміння та кодування.

2. Хороша амортизована продуктивність:

    - `Pairing-Heap-Insert`: $O(1)$
    - `Pairing-Heap-Minimum`: $O(1)$
    - `Pairing-Heap-Meld` (об'єднання двох куп): $O(1)$
    - `Pairing-Heap-Extract-Min`: $O(\log n)$ амортизовано.
    - `Pairing-Heap-Decrease-Key`: $O(\log n)$ амортизовано. Хоча теоретично це $O(\log n)$, на практиці вона часто поводиться як $O(1)$ або $O(\log\log n)$.

- **`Pairing-Heap-Delete`**: $O(\log n)$ амортизовано. Ці характеристики роблять їх дуже швидкими для багатьох реальних застосувань.

3. Ефективність об'єднання (**`Meld`**): Операція об'єднання двох куп є надзвичайно швидкою ($O(1)$), що робить Подвійної купи особливо корисними в алгоритмах, які часто об'єднують пріоритетні черги.

4. Низькі приховані константи: Порівняно з купами Фібоначчі, Подвійні купи мають менші приховані константи в їхній часовій складності, що часто робить їх швидшими на практиці, навіть якщо асимптотична складність не завжди "краща" за Фібоначчі для всіх операцій.

Недоліки

1. Дещо гірша теоретична продуктивність для **`Decrease-Key`**: Хоча на практиці **`Decrease-Key`** працює дуже добре, її амортизована часова складність у найгіршому випадку не така хороша, як $O(1)$ у Купах Фібоначчі. Це може бути проблемою в дуже специфічних теоретичних сценаріях, де послідовність операцій є "злісною".

2. Негарантована найгірша часова складність: Амортизований аналіз куп подвійна є складним, і точні межі для деяких операцій (особливо **`Decrease-Key`**) були предметом досліджень і суперечок. Хоча практична продуктивність чудова, теоретичні гарантії не такі сильні, як у куп Фібоначчі.

3. Вищі накладні витрати на пам'ять, ніж у бінарних куп: Кожен вузол все ще потребує кількох посилань (**`child`**, **`left_sibling`**, **`right_sibling`**, **`parent`**), що робить їх більш пам'ятезалежними, ніж прості бінарні купи.

Подвійні купи є чудовим компромісом між простотою реалізації та високою продуктивністю. Для більшості практичних застосувань, де потрібна ефективна пріоритетна черга з операціями **`Decrease-Key`** та **`Meld`**, Подвійні купи є кращим вибором, ніж купи Фібоначчі, через їхню простоту та менші константи.

Контрольні Запитання

1. У чому полягає основна відмінність подвійної купи від купи Фібоначчі за структурою?

2. Яка операція є найпростішою в купі і чому?
3. Опишіть роль функції `Meld` у подвійній купі.
4. Які два проходи виконуються в операції `Two-Pass-Meld` і навіщо?
5. Як подвійна купа реалізує операцію `Decrease-Key`?
6. Чому подвійна купа часто є кращим практичним вибором, ніж купа Фібоначчі, незважаючи на деякі гірші теоретичні показники?
7. Для яких алгоритмів на графах подвійна купа є ефективним вибором?
8. Які посилання має кожен вузол у подвійній купі?
9. Чи може подвійна купа складатися з кількох окремих дерев? Поясніть.
10. Яка операція в подвійній купі може бути найдорожчою, якщо розглядати її як окрему операцію (не амортизовано)?

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота “Реалізація та аналіз подвійної купи ”.

2. Мета

Метою даної лабораторної роботи є закріплення теоретичних знань про структуру та операції подвійної купи, а також набуття практичних навичок у її реалізації та тестуванні. Робота спрямована на глибоке розуміння балансу між простотою реалізації та амортизованою часовою складністю, що є характерним для цієї структури даних.

3. Завдання
4. Реалізувати структуру даних “Подвійна купа” з усіма основними операціями:
    - `Make-Pairing-Heap()`
    - `Pairing-Heap-Insert(H, x)`
    - `Pairing-Heap-Minimum(H)`

- `Pairing-Heap-Extract-Min(H)`
- `Pairing-Heap-Meld(H1, H2)`
- `Pairing-Heap-Decrease-Key(H, x, k)`
- `Pairing-Heap-Delete(H, x)`

5. Реалізувати допоміжні функції: `Meld(H1, H2)` та `Two-Pass-Meld(list_of_nodes)`.
6. Розробити тестовий набір для перевірки коректності реалізації кожної операції.
7. Продемонструвати роботу подвійної купи на прикладі однієї з її областей застосування (наприклад, Алгоритм Прима).
8. Опис

Подвійна купа є одним деревом, що підтримує властивості міні-купи. Кожен вузол містить ключ, а також посилання на батька, першого дочірнього вузла та лівого/правого братів у двозв'язному списку. Ключовими аспектами реалізації є коректне керування цими посиланнями, особливо під час операцій `Meld` та `Two-Pass-Meld`, які формують та перебудовують дерево. Зверніть увагу на правильну обробку граничних випадків (порожня купа, купа з одним елементом). Для операцій `Decrease-Key` та `Delete` необхідно мати посилання на сам вузол, який потрібно модифікувати.

9. Порядок Виконання

a) Визначення структури вузла: Створіть клас або структуру `PairingHeapNode` (або аналогічну назву) з полями `key`, `parent`, `child`, `left_sibling`, `right_sibling`.

b) Визначення структури купи: Створіть клас або структуру `PairingHeap` (або аналогічну назву) з полем `root` (або `H.root`).

c) Реалізація `Make-Pairing-Heap()`: Ініціалізуйте `root` як `NULL`.

d) Реалізація `Meld(H1, H2)`: Ця функція є основою. Реалізуйте логіку об'єднання двох дерев, враховуючи, яке з них має менший корінь і як правильно оновити посилання.

e) Реалізація `Pairing-Heap-Insert(H, x)`: Використайте `Meld` для об'єднання нового вузла з поточним коренем.

f) Реалізація `Pairing-Heap-Minimum(H)`: Поверніть `H.root`.

g) Реалізація `Two-Pass-Meld(list_of_nodes)`: Реалізуйте двопрохідний алгоритм об'єднання списку вузлів. Зверніть увагу на правильну обробку пар та непарної кількості вузлів.

h) Реалізація `Pairing-Heap-Extract-Min(H)`: Реалізуйте кроки з видаленням кореня та викликом `Two-Pass-Meld` для дочірніх вузлів.

i) Реалізація `Pairing-Heap-Meld(H1, H2)` (як зовнішня операція): Використайте вашу внутрішню `Meld` для об'єднання коренів двох куп.

j) Реалізація `Pairing-Heap-Decrease-Key(H, x, k)`: Реалізуйте логіку видалення `x` з його батьківського списку та подальшого об'єднання `x` з коренем купи за допомогою `Meld`.

k) Реалізація `Pairing-Heap-Delete(H, x)`: Використайте `Decrease-Key` (до `MIN_INT`) та `Extract-Min`.

10. Тестування:

- Створіть набір тестових випадків для кожної операції, що охоплюють різні сценарії (порожня купа, купа з одним елементом, складні сценарії об'єднання та зменшення ключа).
- Використайте функцію обходу дерева (наприклад, прямий обхід) для візуалізації структури купи після ключових операцій.

11. Демонстрація застосування: Реалізуйте спрощений приклад Алгоритму Прима або Дейкстри, де подвійна купа використовується як пріоритетна черга. Зробіть висновок щодо її ефективності.

12. Результат

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де реалізована структура даних "Подвійна купа" з усіма описаними операціями та допоміжними функціями, а також приклади тестування кожної операції. Код повинен бути

чистим, добре коментованим та відповідати принципам об’єктно-орієнтованого програмування.

2. Продемонструвати роботу додатку: показати виконання ключових операцій (`Insert`, `Extract-Min`, `Decrease-Key`, `Meld`, `Delete`) на конкретних прикладах, а також роботу алгоритму Прима/Дейкстри з використанням реалізованої подвійної купи. Візуалізація структури купи (наприклад, текстова схема або графічний вивід) після критичних операцій буде перевагою.

3. Бути готовим відповісти на питання:

    - Поясніть детально алгоритм `Two-Pass-Meld`. Чому саме два проходи?
    - Як `Decrease-Key` зберігає амортизовану ефективність, незважаючи на те, що вона може “відрізати” вузол?
    - Опишіть, як подвійна купа керує списками дочірніх вузлів. У чому її відмінність від кругових списків у Фібоначчі?
    - У яких випадках подвійна купа буде кращим вибором, ніж купа Фібоначчі, а коли – гіршим?
    - Продемонструйте на прикладі, як операції `Meld` та `Extract-Min` разом підтримують властивість купи.

Тести

1. Яка операція в Подвійній купі має амортизовану часову складність $O(1)$?

    a) `Extract-Min`

    b) `Decrease-Key`

    c) `Insert`

    d) `Delete`

2. Який з наступних алгоритмів на графах отримує значну вигоду від використання подвійної купи як пріоритетної черги?

    a) Алгоритм Флойда-Воршалла

b) Алгоритм Крускала

c) Алгоритм Прима

d) Алгоритм пошуку в глибину (DFS)

3. Скільки дерев зазвичай містить Купа Подвійна?

   a) Багато

   b) Завжди одне

   c) Залежить від кількості елементів

   d) До лог (n)

4. Яка допоміжна функція використовується для об’єднання двох довільних дерев у подвійній купі?

   a) `Consolidate`

   b) `Cut`

   c) `Meld`

   d) `Link`

5. Що відбувається з вузлом, який вилучається з подвійної купи за допомогою `Extract-Min`?

   a) Його ключ збільшується до нескінченності.

   b) Його дочірні вузли видаляються.

   c) Його дочірні вузли об’єднуються, щоб сформувати новий корінь.

   d) Він стає батьком для всіх інших вузлів.

Відповіді: 1. – c), 2. – c), 3. – b), 4. – c), 5. – c).

Тести підвищеної складності

1. Розгляньте послідовність n операцій `Insert`, а потім n операцій `Extract-Min`. Яка амортизована часова складність цієї послідовності для подвійної купи ?

   a) $O(n)$

b) $O(n\log n)$

c) $O(n + n\log n)$

d) $O(n \cdot \text{ам. } \log n)$

2. Чому операція **`Pairing-Heap-Decrease-Key`** переміщує зменшений вузол до кореня купи, якщо властивість купи порушена?

   a) Для зменшення висоти дерева.

   b) Щоб спростити операцію **`Extract-Min`**.

   c) Щоб зберегти амортизовану ефективність, відкладаючи складніші перебудови.

   d) Тому що тільки кореневий вузол може мати найменший ключ.

3. Яке твердження про **`Two-Pass-Meld`** є вірним?

   a) Воно використовується тільки для операції **`Insert`**.

   b) Воно об’єднує всі вузли в один список за один прохід.

   c) Перший прохід збирає пари вузлів, другий прохід зливає результати пар.

   d) Воно завжди створює бінарне дерево.

4. Яка з наступних переваг є основною причиною використання подвійної купи над Купою Фібоначчі в багатьох практичних застосуваннях?

   a) Краща асимптотична складність для **`Decrease-Key`**.

   b) Менші накладні витрати на пам’ять.

   c) Значно простіша реалізація.

   d) Гарантована найгірша часова складність для всіх операцій.

5. Розгляньте вузол x у подвійній купі. Його дочірні вузли $c_1, c_2, \ldots, c_k$ зберігаються у двозв’язному списку. Якщо **`x.child`** вказує на c_1, на що вказуватиме **`c_k.right_sibling`**?

   a) **`x`**

b) `NULL`

c) `c_1`

d) `c_{k-1}`

6. Яка амортизована часова складність операції `Delete(H, x)` у подвійній купі?

   a) $O(1)$

   b) $O(\log n)$

   c) $O(n)$

   d) $O(n \log n)$

7. Який з наведених механізмів відсутній у подвійній купі, але є ключовим у купі Фібоначчі?

   e) Двозв'язні списки

   f) Посилання на батька

   g) Механізм "позначки" (`marked`)

   h) Об'єднання дерев

8. Якщо `Pairing-Heap-Meld(H1, H2)` об'єднує дві купи, якій з куп буде присвоєно результат об'єднання?

   a) Новій, порожній купі.

   b) Купі з меншим коренем.

   c) Завжди H1.

   d) Завжди H2.

Відповіді на тести підвищеної складності

1. b) n операцій `Insert` займуть $n \times O(1) = O(n)$. n операцій `Extract-Min` займуть $n \times O(\log n) = O(n \log n)$. Загальна амортизована складність є сумою, тобто $O(n + n \log n)$, що спрощується до $O(n \log n)$.

2. c) Переміщення вузла до кореня є швидкою операцією. Складніші перебудови відкладаються до `Extract-Min`, де вони “оплачуються” амортизовано.

3. c) `Two-Pass-Meld` стратегічно об’єднує дочірні вузли у два етапи, щоб ефективно зменшити їх кількість до одного кореня.

4. c) Простота реалізації подвійної купи робить її більш привабливою для більшості практичних застосувань, незважаючи на те, що теоретично `Decrease-Key` у Фібоначчі краще.

5. b) Дочірні вузли подвійної купи організовані у двозв’язний список, який не є круговим. Тому `right_sibling` останнього вузла в списку буде `NULL`.

6. b) Операція `Delete` складається з `Decrease-Key` (яка є амортизовано $O(\log n)$) та `Extract-Min` (також амортизовано $O(\log n)$), отже, загальна амортизована складність $O(\log n)$.

7. c) Механізм `marked` та каскадних вирізань є специфічним для куп Фібоначчі і відсутній у подвійних купах.

8. c) У стандартній реалізації `Pairing-Heap-Meld(H1, H2)`, результат об’єднання зберігається у `H1`, а `H2` робиться порожньою.

Завдання для самостійної роботи

1. Дослідження амортизованого аналізу: Глибше вивчіть, як саме амортизований аналіз застосовується до подвійної купи. Зосередьтеся на тому, як витрати на операцію `Extract-Min` “розподіляються” між попередніми операціями.

2. Порівняння з іншими купами: Проаналізуйте, у яких саме сценаріях (наприклад, щільність графа, частота операцій `Decrease-Key`) подвійна купа перевершує бінарну купу, і коли купа Фібоначчі може бути кращою.

3. Альтернативні реалізації `Two-Pass-Meld`: Дослідіть інші можливі стратегії реалізації `Two-Pass-Meld` або її варіації. Чи існують підходи з одним проходом, і як вони впливають на ефективність?

4. Реалізація Алгоритму Дейкстри: Реалізуйте алгоритм Дейкстри для пошуку найкоротшого шляху, використовуючи вашу реалізацію

подвійної купи як пріоритетну чергу. Порівняйте продуктивність з використанням стандартної бібліотечної пріоритетної черги (якщо доступно).

5. Реалізація алгоритму Прима: Реалізуйте алгоритм Прима для пошуку мінімального остовного дерева, використовуючи вашу реалізацію подвійної купи. Проаналізуйте продуктивність на різних типах графів (розріджені/щільні).
6. Модифікація **`Delete`**: Розгляньте, як можна було б оптимізувати операцію **`Delete(H, x)`** без зменшення ключа до негативної нескінченності. Наприклад, шляхом “вирізання” вузла та подальшого об’єднання його дочірніх вузлів. Як це вплине на складність?
7. Вимірювання продуктивності: Напишіть бенчмарк для вимірювання часу виконання кожної операції подвійної купи для різних розмірів купи ($n = 100,1000,10000,100000$). Порівняйте з теоретичними амортизованими показниками.
8. Візуалізація дерева: Створіть візуалізацію (графічну або текстову) дерева подвійної купи після виконання основних операцій (**`Insert`**, **`Extract-Min`**, **`Meld`**, **`Decrease-Key`**). Це допоможе краще зрозуміти динамічні зміни структури.
9. Оптимізація пам’яті: Проаналізуйте накладні витрати на пам’ять для подвійної купи. Чи існують техніки, які можуть зменшити використання пам’яті без суттєвої втрати продуктивності?
10. Застосування у практичних проектах: Знайдіть приклади реальних проектів або бібліотек, які використовують подвійної купи. Обґрунтуйте, чому саме ця структура даних була обрана для їхніх потреб.

Купа лівостороння

На відміну від бінарних куп, які прагнуть до збалансованості за висотою або кількістю вузлів, лівосторонні купи використовують зовсім інший підхід для підтримки своєї ефективності, а саме — властивість лівосторонності. Ця властивість робить їх особливо ефективними для операції об’єднання (meld), яка є фундаментальною для багатьох інших операцій.

Купа лівостороння — це міні-купа на основі бінарного дерева, що означає, що кожен вузол у дереві має значення, менше або рівне значенню своїх дочірніх вузлів. Ключовою особливістю лівосторонньої купи, яка відрізняє її від інших бінарних куп, є підтримання властивості лівосторонності.

Кожен вузол у лівосторонній купі містить наступну інформацію:

- **`ключ`** (key): Значення вузла, за яким він упорядковується в купі (наприклад, пріоритет).
- **`лівий_дочірній`** (left_child): Посилання на лівого дочірнього вузла.
- **`правий_дочірній`** (right_child): Посилання на правого дочірнього вузла.
- **`npl`** (null path length / довжина нульового шляху): Це найважливіший атрибут. **`npl`** вузла x визначається як довжина найкоротшого шляху від x до вузла, який не має одного або обох своїх дочірніх вузлів (NULL-посилання). Для **`NULL`**-вузла (порожнього посилання) **`npl`** вважається -1. Для вузла, який є листом (не має дочірніх вузлів), його **`npl`** дорівнює 0. Значення **`npl`** для будь-якого вузла x обчислюється як 1 + min(npl(x.left_child),npl(x.right_child)).

Властивість лівосторонності: Для кожного вузла x у лівосторонній купі, **`npl(x.left_child)`** завжди має бути більшим або дорівнювати **`npl(x.right_child)`**.

Тобто:

$$\text{npl(x.left\_child)} \geq \text{npl(x.right\_child)}$$

Ця властивість гарантує, що найкоротший шлях до **`NULL`**-посилання завжди знаходиться у правому піддереві. Це означає, що праве піддерево завжди “легше” або “коротше”, ніж ліве, з точки зору **`npl`**.

Ця асиметрія є ключовою для ефективності операції об’єднання, оскільки вона дозволяє виконувати більшість роботи на коротшому шляху, зберігаючи при цьому загальну логарифмічну висоту купи.

Операції

Лівосторонні купи підтримують такі операції, багато з яких мають логарифмічну часову складність у найгіршому випадку:

1. **`Make-Leftist-Heap()`**: Створення порожньої купи
2. **`Leftist-Heap-Insert(H, x)`**: Вставка нового вузла

3. `Leftist-Heap-Minimum(H)`: Пошук мінімального елемента
4. `Leftist-Heap-Extract-Min(H)`: Видалення мінімального елемента
5. `Leftist-Heap-Meld(H1, H2)`: Об'єднання двох куп
6. `Leftist-Heap-Decrease-Key(H, x, k)`: Зменшення ключа вузла
7. `Leftist-Heap-Delete(H, x)`: Видалення вузла

Давайте детально розглянемо кожну з цих операцій.

Створення порожньої купи `Make-Leftist-Heap()`

Ця операція створює та повертає нову, порожню лівосторонню купу. Вона просто ініціалізує посилання на корінь купи.

Опис: Лівостороння купа `H` може бути представлена як посилання на її кореневий вузол. Якщо купа порожня, це посилання буде `NULL`.

Уявіть порожню площину. Це ваша лівостороння купа. На ній ще немає жодних вузлів.

Купа лівостороння H
H.root = NULL

Алгоритм:

```
Make-Leftist-Heap()
  H.root = NULL  // Ініціалізуємо посилання на кореневий вузол як NUL
L
  Повернути H     // Повертаємо порожню купу
```

Псевдокод:

```
FUNCTION Make-Leftist-Heap():
  H = NEW LeftistHeapObject
  H.root = NIL
  RETURN H
```

Вставка нового вузла `Leftist-Heap-Insert(H, x)`

Ця операція додає новий вузол `x` до Купи Лівосторонньої `H`. Вона виконується за допомогою ключової операції `Meld`.

Опис: Для вставки нового вузла `x` необхідно виконати наступні кроки:

1. Ініціалізація вузла: Створити новий вузол `x` з заданим ключем. Всі його посилання на дочірні елементи і `npl` ініціалізуються:
    - `x.left_child = NULL`
    - `x.right_child = NULL`
    - `x.npl = 0` (новий вузол є листом, тому довжина його нульового шляху дорівнює 0)
2. Об'єднання: Новий вузол `x` (який сам по собі є Лівосторонньою Купою з одним елементом) об'єднується з поточною Купою `H` за допомогою функції `Leftist-Heap-Meld`. Результат цього об'єднання стає новим коренем `H`.

Припустимо, у нас є купа `H` з коренем `A` (ключ 10). Ми хочемо вставити новий вузол `B` (ключ 5).

До вставки:

```
H.root -> (A, 10, npl=...)
           /  \
          ...
```

Після вставки B (використовуючи Meld):

Оскільки ключ B менший за ключ A, B стає новим коренем, а A стає дочірнім вузлом B (Meld буде визначено пізніше, але в результаті B буде коренем).

```
H.root -> (B, 5, npl=...)
           / \
         NIL (A, 10, npl=...)
             / \
            ...
```

Алгоритм:

```
Leftist-Heap-Insert(H, x)
  // Ініціалізація нового вузла
  x.left_child = NULL
  x.right_child = NULL
  x.npl = 0

  // Об'єднати H з новим вузлом (як окремою купою )
  H.root = Leftist-Heap-Meld(H.root, x)
```

Псевдокод:

```
FUNCTION Leftist-Heap-Insert(H, x):
  // Ініціалізація вузла x
  x.left_child = NIL
  x.right_child = NIL
  x.npl = 0

  // Об'єднати H з новим вузлом x
  H.root = Leftist-Heap-Meld(H.root, x)
```

Пошук мінімального елемента `Leftist-Heap-Minimum(H)`

Ця операція повертає вузол з найменшим ключем у Купі Лівосторонній `H`. Завдяки властивості міні-купи, мінімальний елемент завжди є коренем дерева.

Опис: Купа лівостороння зберігає посилання на кореневий вузол у полі `H.root`. Оскільки корінь завжди має найменший ключ серед усіх вузлів у купі, достатньо просто повернути значення цього поля. Ця операція займає константний час $O(1)$.

Припустимо, у нас є купа `H` з коренем `B`.

```
  H.root -> (B, 5, npl=...)  (мінімальний)
             / \
            ...
```

Алгоритм:

```
Leftist-Heap-Minimum(H)
  Повернути H.root
```

Псевдокод:

```
FUNCTION Leftist-Heap-Minimum(H):
  RETURN H.root
```

Вилучення мінімального елемента `Leftist-Heap-Extract-Min(H)`

Ця операція видаляє вузол з найменшим ключем (корінь) з Купи Лівосторонньої `H` і повертає його. Це одна з основних операцій, яка також використовує `Meld`.

Опис: Вилучення мінімального елемента включає наступні кроки:

1. Збереження кореня: Зберегти посилання на поточний кореневий вузол `z = H.root`.

2. Перевірка порожньої купи: Якщо `z` є `NULL`, купа порожня, повертаємо `NULL`.
3. Оновлення кореня: Новий корінь купи утворюється шляхом об'єднання лівого та правого дочірніх вузлів видаленого кореня `z`.
    - `H.root = Leftist-Heap-Meld(z.left_child, z.right_child)`.
4. Очищення посилань: Встановити `z.left_child = NULL` та `z.right_child = NULL` для видаленого вузла `z`, щоб він був повністю від'єднаний від купи.

Нехай до виконання `Extract-Min`:

```
H.root -> (A, 5, npl=2)
           /           \
          (B, 10, npl=1) (C, 15, npl=0)
         /   \           / \
        (D, 20, npl=0) (E, 25, npl=0)
```

Тоді після `Extract-Min`:

– Видаляємо `A`.

– Об'єднуємо `B` та `C`. Припустимо, `B.key < C.key`, тому `B` стає коренем.

– `C` стає правим дочірнім `B`.

```
H.root -> (B, 10, npl=1)
           /           \
          (D, 20, npl=0) (C, 15, npl=0)
                         / \
                       NIL NIL
```

Алгоритм `Leftist-Heap-Extract-Min(H)`:

```
Leftist-Heap-Extract-Min(H)
  z = H.root
  Якщо z == NULL:
    Повернути NULL

  H.root = Leftist-Heap-Meld(z.left_child, z.right_child) // Об'єднує
мо дочірні вузли видаленого кореня

  // Очистити посилання з видаленого вузла для коректного повернення
  z.left_child = NULL
```

```
  z.right_child = NULL

  Повернути z
```

Псевдокод `Leftist-Heap-Extract-Min(H)` такий:

```
FUNCTION Leftist-Heap-Extract-Min(H):
  z = H.root
  IF z IS NIL:
    RETURN NIL

  H.root = Leftist-Heap-Meld(z.left_child, z.right_child)

  // Очистити посилання з видаленого вузла
  z.left_child = NIL
  z.right_child = NIL

  RETURN z
```

Об’єднання двох куп `Leftist-Heap-Meld(H1, H2)`

Ця операція об’єднує дві лівосторонні купи **H1** та **H2** в одну нову купу. Це єдина фундаментальна операція, на якій базуються всі інші. Вона рекурсивно об’єднує купи, підтримуючи властивість міні-купи та властивість лівосторонності.

Опис: Функція **Meld** приймає два корені дерев **node1** та **node2** (які можуть бути **NULL**) і повертає корінь нового об’єднаного дерева.

1. Базові випадки:
    - Якщо **node1** є **NULL**, повертаємо **node2**.
    - Якщо **node2** є **NULL**, повертаємо **node1**.
2. Визначення кореня: Порівняти ключі **node1.key** та **node2.key**. Вузол з меншим ключем стає коренем нового дерева. Нехай це буде **root_node**, а інший вузол — **other_node**.
    - Якщо **node1.key <= node2.key**: **root_node = node1**, **other_node = node2**.
    - Інакше: **root_node = node2**, **other_node = node1**.

3. Рекурсивне об’єднання: Праве піддерево **root_node** (тобто **root_node.right_child**) рекурсивно об’єднується з **other_node**. Результат цього об’єднання стає новим правим дочірнім вузлом **root_node**.
    - **root_node.right_child = Leftist-Heap-Meld(root_node.right_child, other_node)**.
4. Підтримання властивості лівосторонності: Після рекурсивного об’єднання, необхідно перевірити, чи не порушилася властивість лівосторонності для **root_node**. Якщо **npl(root_node.left_child) < npl(root_node.right_child)**, то лівий і правий дочірні вузли **root_node** міняються місцями. Це гарантує, що ліве піддерево завжди має **npl** не менший, ніж праве.
5. Оновлення **npl**: Оновити **npl** для **root_node** за формулою: 1 + min(npl(root_node.left_child),npl(root_node.right_child)).

Допоміжна функція **get_npl(node)**: Ця функція повертає **npl** вузла, обробляючи **NULL**-посилання.

```
get_npl(node)
  Якщо node == NULL:
    Повернути -1
  Інакше:
    Повернути node.npl
```

Візуалізація роботи **Meld**:

Meld(H1, H2)

```
  H1.root -> (A, 5, npl=1)            H2.root -> (F, 6, npl=1)
             /  \                                 / \
            (B, 10, npl=0) (C, 15, npl=0) (G, 12, npl=0) (H, 18, npl=0)
```

Крок 1: A.key (5) < F.key (6).

Тому A - корінь, F - інший вузол. Рекурсивно викликаємо **Meld(A.right_child (C), F)**.

Meld(C, F)

```
  C (15, npl=0)                 F (6, npl=1)
   / \                         / \
  NIL NIL                    (G, 12, npl=0) (H, 18, npl=0)
```

Крок 2: F.key (6) < C.key (15).

Тому F - корінь, C - інший вузол. Рекурсивно викликаємо `Meld(F.right_child (H), C)`.

Meld(H, C)

```
H (18, npl=0)                C (15, npl=0)
 / \                         / \
NIL NIL                    NIL NIL
```

Крок 3: C.key (15) < H.key (18).

Тому C - корінь, H - інший вузол. Рекурсивно викликаємо `Meld(C.right_child (NIL), H)`. Результат `H`.

Повернення з рекурсії:

- `C.right_child = H`. `npl(C.left_child)` (NIL, -1) vs `npl(C.right_child)` (H, 0). `npl(left) < npl(right)` - міняємо місцями.
  - `C.left_child` стає `H`, `C.right_child` стає `NIL`. `C.npl` = 1 + $\min(-1, -1) = 0$.
  - Повертаємо `C`.
- `F.right_child = C`. `npl(F.left_child)` (G, 0) vs `npl(F.right_child)` (C, 0). Не міняємо. `F.npl` = 1 + $\min(0,0) = 1$.
  - Повертаємо `F`.
- `A.right_child = F`. `npl(A.left_child)` (B, 0) vs `npl(A.right_child)` (F, 1). `npl(left) < npl(right)` - міняємо місцями.
  - `A.left_child` стає `F`, `A.right_child` стає `B`. `A.npl` = 1 + $\min(1,0) = 1$.
  - Повертаємо `A`.

Остаточний результат Meld(H1, H2):

```
H.root -> (A, 5, npl=1)
            /          \
          (F, 6, npl=1) (B, 10, npl=0)
         / \
        (G, 12, npl=0) (C, 15, npl=0)
```

```
 / \
(H, 18, npl=0) NIL
```

Алгоритм `Leftist-Heap-Meld(H1_node, H2_node)`:

```
Leftist-Heap-Meld(H1_node, H2_node)
  // Базові випадки: якщо одна з куп порожня
  Якщо H1_node == NULL:
    Повернути H2_node
  Якщо H2_node == NULL:
    Повернути H1_node

  // Визначити корінь об'єднаної купи (вузол з меншим ключем)
  Якщо H1_node.key > H2_node.key:
    SWAP(H1_node, H2_node) // Забезпечуємо, що H1_node завжди має мен
ший ключ

  // Рекурсивно об'єднати праве піддерево кореня з іншою купою
  H1_node.right_child = Leftist-Heap-Meld(H1_node.right_child, H2_nod
e)

  // Підтримка властивості лівосторонності:
  // Якщо npl лівого дочірнього менше, ніж npl правого дочірнього, по
міняти їх місцями
  Якщо get_npl(H1_node.left_child) < get_npl(H1_node.right_child):
    SWAP(H1_node.left_child, H1_node.right_child)

  // Оновити npl кореня
  H1_node.npl = 1 + get_npl(H1_node.right_child)

  Повернути H1_node
```

Псевдокод `Leftist-Heap-Meld(H1_node, H2_node)` такий:

```
FUNCTION Leftist-Heap-Meld(H1_node, H2_node):
  // Базові випадки
  IF H1_node IS NIL:
    RETURN H2_node
  IF H2_node IS NIL:
    RETURN H1_node

  // Переконатися, що H1_node має менший ключ
  IF H1_node.key > H2_node.key:
    SWAP(H1_node, H2_node)
```

```
  // Рекурсивно об'єднати праве піддерево H1_node з H2_node
  H1_node.right_child = Leftist-Heap-Meld(H1_node.right_child, H2_nod
e)

  // Перевірка та відновлення властивості лівосторонності
  IF get_npl(H1_node.left_child) < get_npl(H1_node.right_child):
    SWAP(H1_node.left_child, H1_node.right_child)

  // Оновлення npl для H1_node
  H1_node.npl = 1 + get_npl(H1_node.right_child)

  RETURN H1_node
```

Псевдокод `get_npl(node)`:

```
FUNCTION get_npl(node):
  IF node IS NIL:
    RETURN -1
  ELSE:
    RETURN node.npl
```

Зменшення ключа вузла `Leftist-Heap-Decrease-Key(H, x, k)`

Ця операція зменшує ключ вузла `x` до нового значення `k`. Важливо, що `k` повинно бути меншим за поточний ключ `x.key`.

Якщо після зменшення ключа властивість міні-купи порушується (тобто `x.key < x.parent.key`), то вузол `x` від'єднується від його батька і об'єднується з коренем купи.

Цей процес також вимагає перебалансування шляху від батька `x` до кореня купи.

Опис:

1. Перевірка вхідних даних:
    - Перевіряється, чи новий ключ `k` дійсно менший за поточний ключ `x.key`. Якщо ні, це є помилкою, оскільки купа не підтримує збільшення ключів.
    - Якщо вузол `x` вже є коренем купи (`x == H.root`), або якщо його новий ключ `k` все ще більший або дорівнює ключу його батька (`x.parent.key`), то властивість міні-купи не порушується. У цьому випадку просто оновлюється `x.key`, і операція завершується.

2. Оновлення ключа: Якщо попередня перевірка успішна (тобто `x` не є коренем і `k < x.parent.key`), то встановлюється `x.key = k`.
3. Від'єднання вузла `x` від його батька (`x.parent`):
    - Зберігається посилання на батьківський вузол `p = x.parent`.
    - Визначається, чи є `x` лівим (`p.left_child == x`) або правим (`p.right_child == x`) дочірнім вузлом `p`. Відповідне посилання у `p` встановлюється на `NULL`.
    - Вузол `x` повністю від'єднується від батьківського дерева: його посилання `x.parent` стає `NULL`, а його власні дочірні посилання `x.left_child` та `x.right_child` також тимчасово встановлюються на `NULL` (оскільки він стає коренем "нового" одиничного дерева), а його `npl` встановлюється на 0 (як для листа).
4. Перебалансування батьківського шляху (`rebalance_path_up`):
    - Оскільки вузол `p` втратив одного зі своїх дочірніх вузлів, це могло порушити властивість лівосторонності або змінити його `npl`.
    - Необхідно рекурсивно або ітеративно пройти вгору від `p` до кореня купи (`H.root`), перевіряючи та відновлюючи властивість лівосторонності для кожного вузла на цьому шляху. Для кожного вузла `curr` на шляху:
        - Вузол `curr` об'єднує свої поточні ліве та праве піддерева за допомогою `Leftist-Heap-Meld`. Результат цього об'єднання стає новим лівим дочірнім `curr`, а його правий дочірній стає `NULL` (перед наступним обміном).
        - Якщо `npl(curr.left_child) < npl(curr.right_child)`, лівий і правий дочірні вузли `curr` обмінюються місцями.
        - `npl(curr)` оновлюється.
        - Продовжується для `curr.parent`.
5. Об'єднання `x` з основною купою :

- Після того, як шлях від `p` до кореня `H.root` був перебалансований, і `x` став окремим коренем, `x` об’єднується з поточним коренем `H.root` за допомогою `Leftist-Heap-Meld`. Результат цього об’єднання стає новим `H.root`.

Операція `rebalance_path_up` є складною і часто не реалізується явно в спрощених прикладах Leftist Heaps. На практиці `Decrease-Key` іноді реалізується як `Delete` (вилучення) з подальшою `Insert` (вставкою) нового елемента з меншим ключем, що є простішим, але, можливо, менш ефективним у теорії.

Для цього підручника ми надамо псевдокод, який намагається виконати “чисту” `Decrease-Key` з необхідними перебалансуваннями.

Алгоритм `Leftist-Heap-Decrease-Key(H, x, k)`

```
Leftist-Heap-Decrease-Key(H, x, k)
  // 1. Перевірка вхідних даних
  Якщо k > x.key:
    Помилка "Новий ключ більший за поточний"

  // 2. Якщо x є коренем або властивість купи не порушується
  Якщо x == H.root:
    x.key = k
    Повернути

  // Припускаємо, що x.parent існує, якщо x не корінь
  Якщо k >= x.parent.key:
    x.key = k
    Повернути

  // 3. Оновлення ключа та від'єднання x
  x.key = k
  p = x.parent // Батьківський вузол x

  // Від'єднати x від його батька
  Якщо p.left_child == x:
    p.left_child = NULL
  Інакше: // p.right_child == x
    p.right_child = NULL

  // Зробити x окремим, "чистим" вузлом
  x.parent = NULL
```

```
  x.left_child = NULL
  x.right_child = NULL
  x.npl = 0 // Він тепер лист (як корінь одиничного дерева)

  // 4. Перебалансування батьківського шляху (від p вгору до кореня)
  // Цей допоміжний алгоритм `rebalance_path_up` буде описано окремо.
  rebalance_path_up(H, p)

  // 5. Об'єднання x з основною купою
  H.root = Leftist-Heap-Meld(H.root, x)
```

Допоміжний Алгоритм `rebalance_path_up(H, current_node)`

Ця функція рекурсивно проходить вгору по шляху від `current_node` до кореня купи, забезпечуючи, що кожен вузол на цьому шляху залишається валідною лівосторонньою купою після “вирізання” дочірнього вузла.

```
rebalance_path_up(H, current_node)
  // Базовий випадок: якщо досягли кореня (current_node.parent == NUL
L) або NULL
  Якщо current_node == NULL:
    Повернути
  Якщо current_node == H.root: // Якщо ми обробляємо сам корінь, то й
ого перебалансування не впливає на його батька
    // Перебалансувати сам корінь, якщо він втратив дитину
    // Це буде зроблено в Meld, коли H.root буде об'єднано з x.
    // Тут ми просто перевіряємо його npl і міняємо дітей, якщо треба

    // Створення тимчасового об'єднаного дерева з дітей current_node
    temp_merged_children = Leftist-Heap-Meld(current_node.left_child,
current_node.right_child)

    // Присвоєння result до лівого дочірнього
    current_node.left_child = temp_merged_children
    current_node.right_child = NULL // Правий дочірній стає NULL для
майбутніх обмінів

    // ОБОВ'ЯЗКОВИЙ обмін, щоб maintain leftist property
    Якщо get_npl(current_node.left_child) < get_npl(current_node.righ
t_child):
        SWAP(current_node.left_child, current_node.right_child)

    current_node.npl = 1 + get_npl(current_node.right_child)
    Повернути
```

```
  // Рекурсивний виклик для батьківського вузла
  parent_node = current_node.parent

  // Тимчасово від'єднуємо current_node від parent_node
  Якщо parent_node.left_child == current_node:
    parent_node.left_child = NULL
  Інакше:
    parent_node.right_child = NULL

  current_node.parent = NULL // current_node тимчасово без батька

  // Перебалансувати current_node
  // Це об'єднання його лівого та правого піддерев
  temp_merged_children = Leftist-Heap-Meld(current_node.left_child, c
urrent_node.right_child)

  current_node.left_child = temp_merged_children
  current_node.right_child = NULL

  // Якщо temp_merged_children не NULL, встановити його батька як cur
rent_node
  Якщо temp_merged_children != NULL:
    temp_merged_children.parent = current_node

  // Обмін дітей, щоб підтримувати властивість лівосторонності
  Якщо get_npl(current_node.left_child) < get_npl(current_node.right_
child):
      SWAP(current_node.left_child, current_node.right_child)

  current_node.npl = 1 + get_npl(current_node.right_child)

  // Повторно приєднати current_node до його батька (parent_node)
  // Це є ключовим, оскільки ми "просуваємося" вгору

  // Залежить від того, яким дочірнім вузлом був current_node для par
ent_node
  // (ми тимчасово від'єднали його).
  // Тут ми можемо або передати current_node назад у рекурсивному вик
лику,
  // або переоб'єднати його з parent_node.

  // Це складний момент. Найпростіший спосіб: `Meld` parent_node з cu
rrent_node.
  // Але це буде змінювати структуру дерева.
```

```
  // Уникнути складності "переприєднання":
  // Найкращий підхід для `rebalance_path_up` - це зробити її рекурсивною
  // функцією, яка приймає шлях і ребалансує вузли, повертаючи оновлений корінь піддерева.
```

УТОЧНЕНИЙ `rebalance_path_up` АЛГОРИТМ

```
  // Цей алгоритм має працювати "знизу вгору"

  // Вузол, який необхідно ребалансувати після модифікації його дочірніх вузлів
  // (або після того, як один з його дочірніх вузлів був вирізаний)
  rebalance_up(H, current_node) // current_node - це `p` з Decrease-Key

  // Внутрішня функція (або рекурсивний виклик)
  FUNCTION rebalance_up(H, node):
    Якщо node == NULL:
      Повернути

    // Перевірити і, якщо потрібно, обміняти дітей node
    Якщо get_npl(node.left_child) < get_npl(node.right_child):
      SWAP(node.left_child, node.right_child)

    // Оновити npl node
    node.npl = 1 + get_npl(node.right_child)

    // Якщо node не є коренем, продовжити ребалансування вгору
    Якщо node != H.root:
      // Перебалансувати батька node
      rebalance_up(H, node.parent)
```

Отже, остаточний алгоритм `Leftist-Heap-Decrease-Key` з використанням допоміжної `rebalance_up` буде таким:

```
Leftist-Heap-Decrease-Key(H, x, k)
  // 1. Перевірка вхідних даних
  Якщо k > x.key:
    Помилка "Новий ключ більший за поточний"

  // 2. Якщо x є коренем, просто оновлюємо ключ
  Якщо x == H.root:
    x.key = k
    Повернути
```

```
  // 3. Якщо новий ключ не порушує властивість купи, просто оновлюємо
ключ
  // (Припускаємо, що x.parent існує, якщо x не корінь)
  Якщо k >= x.parent.key:
    x.key = k
    Повернути

  // 4. Оновлення ключа
  x.key = k

  p = x.parent // Батьківський вузол x

  // 5. Від'єднання x від його батька
  Якщо p.left_child == x:
    p.left_child = NULL
  Інакше: // p.right_child == x
    p.right_child = NULL

  // Встановлення x як кореня окремого дерева
  x.parent = NULL
  x.left_child = NULL
  x.right_child = NULL
  x.npl = 0

  // 6. Рекурсивне перебалансування шляху від p до кореня
  rebalance_up(H, p)

  // 7. Об'єднання x з основною купою
  H.root = Leftist-Heap-Meld(H.root, x)
```

Псевдокод `Leftist-Heap-Decrease-Key(H, x, k)` такий:

```
FUNCTION Leftist-Heap-Decrease-Key(H, x, k):
  // 1. Перевірка вхідних даних
  IF k > x.key:
    ERROR "New key is greater than current key."

  // 2. Якщо x є коренем, просто оновлюємо ключ
  IF x IS H.root:
    x.key = k
    RETURN

  // 3. Якщо новий ключ не порушує властивість купи
  // (Припускаємо, що x.parent існує, якщо x не корінь)
  IF k >= x.parent.key:
```

```
    x.key = k
    RETURN

  // 4. Оновлення ключа
  x.key = k

  p = x.parent // Батьківський вузол x

  // 5. Від'єднання x від його батька
  IF p.left_child IS x:
    p.left_child = NIL
  ELSE: // p.right_child IS x
    p.right_child = NIL

  // Встановлення x як кореня окремого дерева
  x.parent = NIL
  x.left_child = NIL
  x.right_child = NIL
  x.npl = 0 // Він тепер лист (як корінь одиничного дерева)

  // 6. Рекурсивне перебалансування шляху від p до кореня
  // Цей допоміжний алгоритм `rebalance_up` буде описано нижче.
  rebalance_up(H, p)

  // 7. Об'єднання x з основною купою
  H.root = Leftist-Heap-Meld(H.root, x)
```

Псевдокод `rebalance_up(H, node)` (Допоміжна функція)

Ця функція відповідає за перебалансування вузлів на шляху від зміненого вузла (або його батька) до кореня купи.

```
FUNCTION rebalance_up(H, node):
  IF node IS NIL:
    RETURN

  // Перевірка і, якщо потрібно, обмін дітей `node` для підтримки вла
стивості лівосторонності
  IF get_npl(node.left_child) < get_npl(node.right_child):
    SWAP(node.left_child, node.right_child)

  // Оновлення `npl` для `node`
  node.npl = 1 + get_npl(node.right_child)

  // Якщо `node` не є коренем купи, продовжуємо ребалансування вгору
```

```
  IF node IS NOT H.root:
    // Рекурсивний виклик для батьківського вузла
    // Це забезпечить, що зміни поширюватимуться вгору по дереву.
    rebalance_up(H, node.parent)
```

У реальних реалізаціях **rebalance_up** може бути більш складною, включаючи тимчасові від'єднання та повторні приєднання вузлів до їхніх батьків. Проте, представлений псевдокод відображає основну ідею: після зміни піддерева, необхідно перевірити та оновити **npl** та властивість лівосторонності для батьківського вузла, і так далі, аж до кореня.

Видалення вузла `Leftist-Heap-Delete(H, x)`

Ця операція видаляє довільний вузол **x** з лівосторонньої купи **H**. Це може бути виконано шляхом зменшення ключа **x** до негативної нескінченності, а потім викликом **Extract-Min**.

Опис:

1. Зменшення ключа: Викликати **Leftist-Heap-Decrease-Key(H, x, -infinity)** (або дуже мале число, наприклад, **MIN_INT**). Це гарантує, що **x** стане мінімальним елементом купи.
2. Вилучення мінімуму: Викликати **Leftist-Heap-Extract-Min(H)**. Це видалить **x** (який тепер є мінімальним елементом) з купи.

Візуалізація роботи:

До **Delete(H, G)** (G має ключ 12):

```
H.root -> (A, 5, npl=2)
           /           \
         (B, 10, npl=1) (C, 15, npl=0)
        /   \           / \
      (G, 12, npl=0) (E, 25, npl=0)
```

Крок 1: **Decrease-Key(H, G, -infinity)**

G стає кореневим вузлом (або приєднується до кореня), і **H.root** вказує на **G**.

```
H.root -> (G, -infinity, npl=...)
           / \
         (A, 5, npl=...) (інші вузли A)
```

Крок 2: **Extract-Min(H)**

`G` (який тепер є мінімальним) видаляється. Дочірні вузли `G` обробляються за допомогою `Meld`, щоб сформувати новий корінь купи.

Після `Delete`:

```
  H.root -> (A, 5, npl=...) (залежить від початкового H.root та G.children)
              /           \
             (B, 10, npl=1) (C, 15, npl=0)
            /   \           / \
          NIL   (E, 25, npl=0)
```

Алгоритм:

```
Leftist-Heap-Delete(H, x)
  Leftist-Heap-Decrease-Key(H, x, -infinity) // Зменшуємо ключ до найменшого можливого значення
  Leftist-Heap-Extract-Min(H) // Видаляємо теперішній мінімальний елемент (який є x)
```

Псевдокод:

```
FUNCTION Leftist-Heap-Delete(H, x):
  // Припускаємо, що є константа NEGATIVE_INFINITY
  Leftist-Heap-Decrease-Key(H, x, NEGATIVE_INFINITY)
  Leftist-Heap-Extract-Min(H)
```

Обходи Дерева

Оскільки лівостороння купа є бінарним деревом, до неї застосовуються стандартні алгоритми обходу бінарних дерев: прямий обхід (Pre-order), центрований обхід (In-order), зворотний обхід (Post-order) та обхід у ширину (BFS).

Структура вузла для обходу

```
Структура Вузла:
  key: int
  left_child: Node*
  right_child: Node*
  npl: int
  // parent: Node* (опціонально, але корисно для Decrease-Key/Delete)
```

3. Обхід у глибину (Depth-First Search - DFS)

1.1. Прямий обхід

Відвідати кореневий вузол, потім рекурсивно відвідати ліве піддерево, потім праве піддерево.

Візуалізація роботи:

```
    A
   / \
  B   C
```

```
Порядок обходу: A -> B -> C
```

Алгоритм:

```
Pre-order-Traversal(node)
  Якщо node == NULL:
    Повернути
  Відвідати(node) // Наприклад, вивести node.key

  Pre-order-Traversal(node.left_child)
  Pre-order-Traversal(node.right_child)
```

Псевдокод:

```
PROCEDURE Pre-order-Traversal(node):
  IF node IS NIL:
    RETURN

  VISIT(node) // Наприклад, PRINT node.key

  Pre-order-Traversal(node.left_child)
  Pre-order-Traversal(node.right_child)
```

1.2. Центрований обхід

Опис: Рекурсивно відвідати ліве піддерево, потім відвідати кореневий вузол, потім рекурсивно відвідати праве піддерево.

Візуалізація роботи:

```
    A
   / \
  B   C
```

```
Порядок обходу: B -> A -> C
```

Алгоритм:

```
In-order-Traversal(node)
  Якщо node == NULL:
    Повернути

  In-order-Traversal(node.left_child)
  Відвідати(node) // Наприклад, вивести node.key
  In-order-Traversal(node.right_child)
```

Псевдокод:

```
PROCEDURE In-order-Traversal(node):
  IF node IS NIL:
    RETURN

  In-order-Traversal(node.left_child)
  VISIT(node) // Наприклад, PRINT node.key
  In-order-Traversal(node.right_child)
```

1.3. Зворотний Обхід

Рекурсивно відвідати ліве піддерево, потім праве піддерево, а потім відвідати кореневий вузол.

Візуалізація роботи:

```
    A
   / \
  B   C

Порядок обходу: B -> C -> A
```

Алгоритм:

```
Post-order-Traversal(node)
  Якщо node == NULL:
    Повернути

  Post-order-Traversal(node.left_child)
  Post-order-Traversal(node.right_child)
  Відвідати(node) // Наприклад, вивести node.key
```

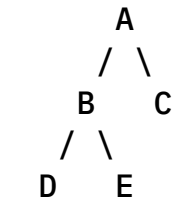
Псевдокод:

```
PROCEDURE Post-order-Traversal(node):
  IF node IS NIL:
    RETURN

  Post-order-Traversal(node.left_child)
  Post-order-Traversal(node.right_child)
  VISIT(node) // Наприклад, PRINT node.key
```

1.4. Обхід у ширину

Відвідати вузли рівень за рівнем, починаючи від кореня. Використовує чергу для відстеження вузлів, які потрібно відвідати.

Візуалізація роботи:




```
Порядок обходу: A -> B -> C -> D -> E
```

Алгоритм:

```
BFS-Traversal(root_node)
  Якщо root_node == NULL:
    Повернути

  queue = Нова_Черга()
  queue.Додати(root_node)

  Поки queue не порожня:
    current_node = queue.Видалити()
    Відвідати(current_node) // Наприклад, вивести current_node.key

    Якщо current_node.left_child != NULL:
      queue.Додати(current_node.left_child)
    Якщо current_node.right_child != NULL:
      queue.Додати(current_node.right_child)
```

Псевдокод:

```
PROCEDURE BFS-Traversal(root_node):
  IF root_node IS NIL:
    RETURN

  queue = NEW QUEUE()
  queue.ENQUEUE(root_node)

  WHILE NOT queue.IS_EMPTY():
    current_node = queue.DEQUEUE()
    VISIT(current_node) // Наприклад, PRINT current_node.key

    IF current_node.left_child IS NOT NIL:
      queue.ENQUEUE(current_node.left_child)
```

```
    IF current_node.right_child IS NOT NIL:
      queue.ENQUEUE(current_node.right_child)
```

Обхід всієї купи

Оскільки купа лівостороння є одним деревом, для обходу всієї купи достатньо викликати відповідну функцію обходу для її кореня:

```
Leftist-Heap-Full-Traversal(H)
  Pre-order-Traversal(H.root) // Або In-order-Traversal, Post-order-T
raversal, BFS-Traversal
```

2. Область Застосування

Лівосторонні купи є відмінним вибором для сценаріїв, де операція об'єднання (meld) двох пріоритетних черг є частою і критично важливою. Їхня структура оптимізована саме для цієї операції, що робить їх ефективними у багатьох алгоритмах.

Основними областями застосування лівосторонніх куп є:

- Алгоритми на графах: Хоча вони не завжди виграють у `Decrease-Key` так само, як Купи Фібоначчі чи Подвійна, їхня ефективність об'єднання робить їх корисними в деяких алгоритмах.
- Симуляції дискретних подій: У системах, де потрібно керувати чергою подій, і де може виникнути потреба об'єднати списки подій.
- Алгоритми обчислення найкоротших шляхів: У варіаціях алгоритму Дейкстри або інших алгоритмах, які можуть потребувати об'єднання пріоритетних черг.
- Алгоритми об'єднання множин (Disjoint Set Union) з оптимізацією: Хоча не прямо, ідея об'єднання лівосторонніх дерев може бути застосована в деяких структурах для швидкого об'єднання множин.
- Реалізація інших складних структур даних: Лівосторонні Купи можуть служити будівельним блоком для більш складних структур даних, які вимагають швидких операцій злиття.

3. Приклади застосування

Давайте розглянемо конкретні приклади алгоритмів, де лівосторонні купи можуть бути застосовані.

3.1. Алгоритм Дейкстри для пошуку найкоротшого шляху (з модифікацією)

Стандартний алгоритм Дейкстри вимагає ефективної операції `Decrease-Key`. Хоча `Decrease-Key` у лівосторонніх купах може бути реалізована через "видалення-та-вставку", що є $O(\log N)$, її переваги розкриваються, коли ми можемо змінити підхід до оновлення відстаней.

Замість прямого `Decrease-Key` для вже існуючого вузла, ми можемо:

- Вставляти нові (менші) копії вузлів: Коли знайдено коротший шлях до вершини v, ми не змінюємо ключ існуючого вузла v у купі. Замість цього, ми вставляємо нову копію вершини v з новим, меншим ключем до купи.
- Ігнорувати старі копії: Коли `Extract-Min` повертає вершину u, ми перевіряємо, чи була вона вже остаточно оброблена (наприклад, використовуючи масив `visited`). Якщо так, ми просто ігноруємо її та продовжуємо `Extract-Min`. Якщо ні, то це дійсно мінімальна вершина, і ми її обробляємо.

Перевага: Цей підхід дозволяє використовувати операції `Insert` та `Extract-Min`, які обидві мають логарифмічну складність у лівосторонній купі. Хоча це збільшує розмір купи (може бути багато "старих" копій вершин), для деяких графів або сценаріїв це може бути прийнятним. Загальна складність залишається $O(E \log V)$, подібною до використання бінарної купи, але з простою реалізацією.

3.2. Об'єднання черг у паралельних алгоритмах або розподілених системах

У розподілених системах або паралельних алгоритмах часто виникає потреба в об'єднанні результатів або черг пріоритетів з різних підсистем.

Ефективна операція `Meld` ($O(\log n)$) робить купи ідеальними для таких сценаріїв. Наприклад:

- Кілька потоків можуть генерувати елементи з пріоритетами в окремі лівосторонні купи.
- Час від часу ці окремі купи можуть бути об'єднані в одну центральну купу за допомогою `Meld`, щоб отримати загальний мінімальний елемент.

Перевага: Швидке та ефективне злиття дозволяє гнучко керувати розподіленими пріоритетними чергами без значних накладних витрат.

Переваги та Недоліки

Лівосторонні купи мають чіткі переваги та недоліки, які визначають їхнє місце серед пріоритетних черг.

Переваги

1. Ефективна операція об'єднання (`Meld`): Це єдина та головна перевага лівосторонніх куп. Операція `Meld` виконується за $O(\log n)$ часу у найгіршому випадку, де n — загальна кількість вузлів в об'єднаних купах. Це дуже ефективно порівняно з іншими простими купами, де об'єднання може займати $O(n)$ часу (наприклад, у звичайній бінарній купі, коли потрібно перевставляти всі елементи).

2. Простота реалізації: Порівняно з купами Фібоначчі або навіть купами подвійна (які мають складну операцію `Two-Pass-Meld`), лівосторонні купи відносно прості в реалізації, особливо операція `Meld`, яка є рекурсивною та елегантною.

3. Гарантована логарифмічна часова складність: Всі основні операції (`Insert`, `Extract-Min`, `Meld`, `Delete`) мають гарантовану часову складність $O(\log n)$ у найгіршому випадку. Це відрізняє їх від куп подвійна, де деякі межі є амортизованими та іноді менш суворими теоретично.

4. Компактність структури: На відміну від куп Фібоначчі, які є сукупністю дерев, лівостороння купа є одним деревом, що може спростити деякі аспекти управління пам'яттю та обходу.

Недоліки

1. Відносно високі константи: Хоча асимптотична складність є логарифмічною, приховані константи можуть бути більшими, ніж у простіших структур, таких як бінарні купи. Це означає, що для невеликих n бінарна купа може бути швидшою.

2. Складність `Decrease-Key`: Як було показано в описі, реалізація `Decrease-Key` "на місці" з збереженням усіх властивостей може бути складною і вимагати посилань на батьків. Якщо `Decrease-Key` реалізується через "видалення-та-вставку", це призводить до подвоєння логарифмічних витрат. Це робить їх менш привабливими для алгоритмів, що сильно покладаються на часті `Decrease-Key` (наприклад, чистий алгоритм Дейкстри).

3. Вищі накладні витрати на пам’ять, ніж у бінарних куп: Кожен вузол потребує додаткового поля `npl` та посилань на лівого та правого дочірнього вузлів, що збільшує пам’ятемісткість порівняно з масивною реалізацією бінарної купи.

Лівосторонні Купи є відмінним вибором, коли операція об’єднання є ключовою та частою. Їхня простота реалізації та гарантована логарифмічна продуктивність роблять їх надійним вибором, особливо коли амортизована продуктивність не є достатньою гарантією, або коли відсутня потреба в надзвичайно швидкій операції `Decrease-Key` (як у купах Фібоначчі).

Контрольні Запитання

1. Яке основне призначення лівосторонньої купи?
2. Поясніть, що таке `npl` (довжина нульового шляху) вузла в лівосторонній купі.
3. Яку властивість повинна задовольняти лівостороння купа для кожного свого вузла?
4. Яка операція є фундаментальною для всіх інших операцій у лівосторонній купі?
5. Як реалізується операція `Insert` у лівосторонній купі?
6. Що відбувається з коренем купи після виконання операції `Extract-Min`?
7. Чому лівостороння купа вважається “лівосторонньою”?
8. Яка асимптотична часова складність операції `Meld` у найгіршому випадку?
9. Наведіть приклад області застосування, де лівостороння купа виявляє свої переваги.
10. Який недолік лівосторонньої купи може бути критичним для алгоритмів, що часто використовують оновлення пріоритетів?

Методичні вказівки до лабораторної роботи:

1. Назва

Лабораторна робота: “Реалізація та аналіз лівосторонньої купи”.

2. Мета

Метою даної лабораторної роботи є закріплення теоретичних знань про структуру та операції лівосторонньої купи, а також набуття практичних навичок у її реалізації та тестуванні. Робота спрямована на глибоке розуміння принципу лівосторонності та ефективності операції об'єднання.

3. Завдання

4. Реалізувати структуру даних "Лівостороння Купа" з усіма основними операціями:
    - `Make-Leftist-Heap()`
    - `Leftist-Heap-Insert(H, x)`
    - `Leftist-Heap-Minimum(H)`
    - `Leftist-Heap-Extract-Min(H)`
    - `Leftist-Heap-Meld(H1, H2)`
    - `Leftist-Heap-Decrease-Key(H, x, k)` (реалізувати через `Delete` + `Insert` для простоти)
    - `Leftist-Heap-Delete(H, x)`
5. Реалізувати допоміжну функцію `get_npl(node)`.
6. Розробити тестовий набір для перевірки коректності реалізації кожної операції.
7. Продемонструвати роботу Лівосторонньої купи на прикладі задачі, що вимагає частого об'єднання черг.
8. Опис

Лівостороння Купа — це бінарне дерево, що підтримує властивості мінікупи та лівосторонності, яка базується на значенні `npl`. Кожен вузол містить ключ, посилання на лівого та правого дочірніх вузлів та `npl`. Реалізація вимагає рекурсивного підходу до `Meld` та уважного керування `npl` і обміном дочірніх вузлів для підтримки лівосторонності. Для `Decrease-Key` та `Delete` найпростіше реалізувати їх через комбінацію інших операцій (`Meld`, `Insert`, `Extract-Min`), що вимагає, щоб вузли мали посилання на батьків.

9. Порядок Виконання

a) Визначення структури вузла: Створіть клас або структуру `LeftistHeapNode` (або аналогічну назву) з полями `key`, `left_child`, `right_child`, `npl`, а також `parent` (для `Decrease-Key` та `Delete`).

b) Визначення структури купи: Створіть клас або структуру `LeftistHeap` (або аналогічну назву) з полем `root` (або `H.root`).

c) Реалізація `Make-Leftist-Heap()`: Ініціалізуйте `root` як `NULL`.

d) Реалізація `get_npl(node)`: Допоміжна функція для безпечного отримання `npl`.

e) Реалізація `Leftist-Heap-Meld(H1_node, H2_node)`: Це центральна функція. Реалізуйте її рекурсивно, враховуючи базові випадки, порівняння ключів, рекурсивне об'єднання правого піддерева та перевірку/відновлення властивості лівосторонності.

f) Реалізація `Leftist-Heap-Insert(H, x)`: Використайте `Meld` для об'єднання купи з новим вузлом.

g) Реалізація `Leftist-Heap-Minimum(H)`: Поверніть `H.root`.

h) Реалізація `Leftist-Heap-Extract-Min(H)`: Використайте `Meld` для об'єднання лівого та правого дочірніх вузлів видаленого кореня.

i) Реалізація `Leftist-Heap-Delete(H, x)`: Реалізуйте її шляхом виклику `Decrease-Key(H, x, -infinity)` та `Extract-Min(H)`. Це вимагатиме, щоб вузол `x` мав посилання на батька.

j) Реалізація `Leftist-Heap-Decrease-Key(H, x, k)`: Для спрощення, реалізуйте її через логіку:

- Якщо `x` — корінь або `k` не порушує властивість купи, просто оновіть `x.key`.
- Інакше: від'єднайте `x` від його батька, зробіть `x` окремим деревом (з `npl=0`), а потім об'єднайте `x` з `H.root` за допомогою `Meld`. Цей підхід забезпечує коректність, хоча й не є "in-place" операцією.

10. Тестування:

- Створіть декілька тестових випадків для кожної операції, що охоплюють різні сценарії (порожня купа, купа з одним елементом, складні сценарії об’єднання, вставки).
- Відображайте `npl` для кожного вузла після операцій, щоб перевірити властивість лівосторонності.
- Використайте функцію обходу дерева (наприклад, прямий обхід) для візуалізації структури купи після ключових операцій.

11. Демонстрація застосування: Реалізуйте сценарій, де кілька пріоритетних черг (наприклад, від різних “джерел”) об’єднуються в одну головну чергу. Наприклад, симуляція подій з можливістю об’єднання черг подій.

12. Результат

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де реалізована структура даних “Лівостороння Купа” з усіма описаними операціями та допоміжними функціями, а також приклади тестування кожної операції. Код повинен бути чистим, добре коментованим та відповідати принципам об’єктно-орієнтованого програмування.
2. Продемонструвати роботу додатку: показати виконання ключових операцій (`Insert`, `Extract-Min`, `Meld`, `Decrease-Key`, `Delete`) на конкретних прикладах, звертаючи увагу на зміни `npl` та структуру дерева. Візуалізація структури купи (наприклад, текстова схема, що показує зв’язки та `npl`) після критичних операцій є обов’язковою.
3. Бути готовим відповісти на питання:
    - Поясніть, як властивість лівосторонності допомагає оптимізувати операцію `Meld`.
    - Чому у `Meld` ми завжди рекурсивно об’єднуємо з правим дочірнім вузлом кореня?
    - Поясніть, як `npl` вузла x обчислюється, і чому `npl` для `NULL` дорівнює -1.

- Опишіть, як реалізовано `Decrease-Key` у вашій програмі та які його обмеження або особливості.
- Які переваги має Лівостороння Купа над бінарною купою , якщо основним завданням є часте об’єднання черг?

Тести

1. Яке поле є унікальним для вузлів Лівосторонньої купи порівняно зі стандартними бінарними купами?
    a) `key`
    b) `left_child`
    c) `right_child`
    d) `npl`
2. Яка з наступних операцій є найбільш ефективною і фундаментальною для Лівосторонньої купи?
    a) `Insert`
    b) `Extract-Min`
    c) `Meld`
    d) `Decrease-Key`
3. Чому ліве піддерево вузла в Лівосторонній Купі завжди має бути “важчим” (більшим або рівним за `npl`) за праве?
    a) Для балансування висоти дерева.
    b) Для прискорення пошуку мінімального елемента.
    c) Щоб операція `Meld` виконувала більшість роботи на коротшому (правому) шляху.
    d) Для зменшення кількості вузлів у купі.
4. Яке значення `npl` має вузол, який не має дочірніх вузлів (лист)?
    a) -1
    b) 0

c) 1

d) Залежить від висоти дерева.

5. Що відбувається, якщо `npl(node.left_child) < npl(node.right_child)` після рекурсивного виклику `Meld`?

   a) Вузол видаляється.

   b) Дочірні вузли міняються місцями.

   c) Нічого не відбувається.

   d) Збільшується `npl` лівого дочірнього вузла.

Відповіді: 1. – d), 2. – c), 3. – c), 4. – b), 5. – b)

Тести підвищеної складності

1. Розгляньте лівосторонню купу з N вузлами. Яка є асимптотична висота цієї купи?

   a) $O(\log N)$

   b) $O(N)$

   c) $O(\sqrt{N})$

   d) $O(N \log N)$

2. Операція `Delete(H, x)` у лівосторонній купі зазвичай реалізується через `Decrease-Key(H, x, -infinity)` і `Extract-Min(H)`. Яка сумарна часова складність цієї послідовності?

   a) $O(1)$

   b) $O(\log N)$

   c) $O(N)$

   d) $O(N \log N)$

3. Якщо дві лівосторонні купи, H_1 і H_2, мають корені з ключами K_1 та K_2 відповідно. Припустимо, $K_1 < K_2$. Який вузол стане коренем об’єднаної купи після `Meld(H1.root, H2.root)`?

   a) $H_2.root$

b) $H_1.root$

c) Будь-який з них, залежить від `npl`.

d) Новий вузол, створений `Meld`.

4. Уявіть, що ви використовуєте лівосторонню купу для алгоритму Дейкстри і застосовуєте стратегію вставки копій вузлів при оновленні відстаней. Який недолік цього підходу?

   a) Збільшення часової складності `Extract-Min`.

   b) Потреба в механізмі для ігнорування “старих” копій.

   c) Збільшення складності реалізації `Meld`.

   d) Збільшення `npl` для всіх вузлів.

5. Припустимо, у вас є Лист (вузол без дітей) у лівосторонній купі. Яке значення його `npl`?

   a) -1

   b) 0

   c) 1

   d) Не визначено

6. Яка з наступних структур даних є найбільш схожою на лівосторонню купу за принципом злиття (merge-based)?

   a) Бінарна купа

   b) Червоно-чорне дерево

   c) Біноміальна купа

   d) купа подвійна

7. Якщо корінь лівосторонньої купи є `NULL`, що поверне операція `Leftist-Heap-Minimum(H)`?

   a) Від’ємну нескінченність.

   b) Помилку.

c) `NULL`.

d) Випадковий вузол.

8. Яка з наступних переваг лівосторонньої купи може бути особливо важливою у системах реального часу, де передбачуваність є критичною?

a) Амортизована часова складність.

b) Низькі приховані константи.

c) Простота реалізації.

d) Гарантована найгірша часова складність для всіх операцій.

Відповіді

1. a) Властивість лівосторонності гарантує, що висота дерева росте логарифмічно відносно кількості вузлів, оскільки “коротка” сторона (права) завжди є найкоротшим шляхом до `NULL`, обмежуючи зростання.

2. b) Кожна з операцій `Decrease-Key` (в реалізації, яка від’єднує та об’єднує) та `Extract-Min` має логарифмічну часову складність у найгіршому випадку.

3. b) Операція `Meld` завжди вибирає вузол з меншим ключем як корінь об’єднаної купи.

4. b) Якщо ми вставляємо нові копії вершин з меншими відстанями, купа може містити декілька записів для однієї вершини. Алгоритм повинен перевіряти, чи не була вершина вже оброблена.

5. b) Для вузла, який не має дочірніх вузлів (лист), `npl` дорівнює $1 + \min(\text{npl}(\text{NIL}),\text{npl}(\text{NIL})) = 1 + \min(-1,-1) = 1 - 1 = 0$.

6. c) Як Лівостороння, так і Біноміальна Купи мають ефективні операції злиття, які є ключовими для їхньої продуктивності.

7. c) Якщо купа порожня, її корінь є `NULL`, і операція `Leftist-Heap-Minimum` просто повертає це значення.

8. d) На відміну від амортизованих структур (як Фібоначчі або подвійна), лівосторонні купи надають гарантовану логарифмічну

складність для кожної окремої операції, що важливо для систем реального часу.

Завдання для самостійної роботи

1. Побудова дерева `npl`: Для наданої лівосторонньої купи (заданої у вигляді ключів і структури), вручну обчисліть `npl` для кожного вузла і перевірте властивість лівосторонності.

2. Покрокове об'єднання: Виконайте покрокове об'єднання двох невеликих лівосторонніх куп (з 3-5 вузлів кожна), демонструючи, як змінюються `npl` та посилання вузлів на кожному етапі рекурсії `Meld`.

3. Порівняння `Decrease-Key`: Детально порівняйте реалізацію `Decrease-Key` у лівосторонній купі (використану у вашій лабораторній роботі) з реалізацією у купі Фібоначчі та купі Подвійна. Обговоріть їхні переваги та недоліки.

4. Застосування у іграх: Дослідіть, як лівосторонні купи або інші пріоритетні черги можуть бути використані в ігрових алгоритмах, наприклад, для пошуку шляхів персонажів або керування чергою подій.

5. Вимірювання продуктивності: Проведіть експерименти з вашою реалізацією. Виміряйте час виконання операцій `Insert`, `Extract-Min`, `Meld` для різних розмірів купи. Порівняйте отримані результати з теоретичними оцінками $O(\log N)$.

6. Візуалізація структури: Розробіть програму, яка графічно (або за допомогою ASCII-графіки) відображає структуру лівосторонньої купи та `npl` кожного вузла після виконання основних операцій.

7. Реалізація Алгоритму Дейкстри: Реалізуйте алгоритм Дейкстри з лівосторонньою купою , використовуючи стратегію "вставки нових копій" при оновленні відстаней. Оцініть вплив цієї стратегії на розмір купи та загальну продуктивність.

8. Імплементація з посиланнями на батьків: Реалізуйте лівосторонню купу таким чином, щоб кожен вузол мав посилання на батька (`parent`). Це дозволить "чистіше" реалізувати операцію `Decrease-Key` (хоча все одно складно). Опишіть переваги та недоліки такого підходу.

9. Сценарій використання `Meld`: Придумайте та реалізуйте сценарій, де часте використання операції `Meld` є ключовим. Наприклад, об’єднання результатів паралельних обчислень, де кожен потік генерує свою власну купу.

10. Порівняння реалізацій: Якщо ви реалізували купи Фібоначчі та Подвійна в попередніх лабораторних роботах, проведіть пряме порівняння продуктивності всіх трьох типів куп на одному й тому ж наборі тестових даних для різних операцій. Зробіть висновки щодо практичної ефективності кожної з них.

Скошена купа

Це ще одна цікава структура даних, яка реалізує концепцію пріоритетної черги і є варіантом бінарної купи. Скошені купи дуже схожі на лівосторонні купи, але їхня відмінність полягає в тому, що вони не підтримують жодної властивості збалансованості (як, наприклад, `npl` у лівосторонніх куп). Натомість, вони покладаються на безумовне перемикання дочірніх вузлів після кожної операції об’єднання, щоб досягти хорошої амортизованої продуктивності.

Скошена купа — це міні-купа на основі бінарного дерева, що означає, що ключ кожного вузла менший або дорівнює ключам усіх його дочірніх вузлів. На відміну від інших типів куп, скошена купа не має жодних додаткових властивостей, які гарантували б її збалансованість за висотою, таких як `npl` або глибина вузлів. Натомість, вона використовує просту, але ефективну стратегію: після кожного об’єднання (meld) лівий і правий дочірні вузли кореня обмінюються місцями.

Цей постійний обмін дочірніми елементами допомагає “випадковим чином” підтримувати баланс купи в амортизованому сенсі, запобігаючи дегенерації до довгого, тонкого дерева в більшості випадків.

Кожен вузол у скошеній купі містить наступну інформацію:

- **`ключ`** (key): Значення вузла, за яким він упорядковується в купі (наприклад, пріоритет).
- **`лівий_дочірній`** (left_child): Посилання на лівого дочірнього вузла.
- **`правий_дочірній`** (right_child): Посилання на правого дочірнього вузла.

Завдяки відсутності додаткових полів для збалансованості (наприклад, `npl`), скошені купи є дуже компактними та легкими в реалізації.

Операції скошеної купи

Скошені купи підтримують стандартні операції пріоритетної черги, при цьому багато з них мають логарифмічну амортизовану часову складність:

1. `Make-Skew-Heap()`: Створення порожньої скошеної купи
2. `Skew-Heap-Insert(H, x)`: Вставка нового вузла
3. `Skew-Heap-Minimum(H)`: Пошук мінімального елемента
4. `Skew-Heap-Extract-Min(H)`: Вилучення мінімального елемента
5. `Skew-Heap-Meld(H1, H2)`: Об'єднання двох Скошених Куп
6. `Skew-Heap-Decrease-Key(H, x, k)`: Зменшення ключа вузла
7. `Skew-Heap-Delete(H, x)`: Видалення вузла

Давайте детально розглянемо кожну з цих операцій.

Створення порожньої скошеної купи `Make-Skew-Heap()`

Ця операція створює та повертає нову, порожню скошену купу. Вона просто ініціалізує посилання на корінь купи.

Опис: скошена купа `H` може бути представлена як посилання на її кореневий вузол. Якщо купа порожня, це посилання буде `NULL`.

Візуалізація роботи: уявіть порожню площину. Це ваша скошена купа. На ній ще немає жодних вузлів.

скошена купа H
H.root = NULL

Алгоритм:

```
Make-Skew-Heap()
  H.root = NULL  // Ініціалізуємо посилання на кореневий вузол як NUL
L
  Повернути H     // Повертаємо порожню купу
```

Псевдокод:

```
FUNCTION Make-Skew-Heap():
  H = NEW SkewHeapObject
  H.root = NIL
  RETURN H
```

Вставка нового вузла `Skew-Heap-Insert(H, x)`

Ця операція додає новий вузол `x` до скошеної купи `H`. Вона виконується за допомогою ключової операції `Meld`.

Опис: Для вставки нового вузла `x` необхідно виконати наступні кроки:

1. Ініціалізація вузла: Створити новий вузол `x` з заданим ключем. Всі його посилання на дочірні елементи ініціалізуються як `NULL`.
    - `x.left_child = NULL`
    - `x.right_child = NULL`
2. Об’єднання: Новий вузол `x` (який сам по собі є скошеною упою з одним елементом) об’єднується з поточною скошеною купою `H` за допомогою функції `Skew-Heap-Meld`. Результат цього об’єднання стає новим коренем `H`.

Візуалізація роботи:

Припустимо, у нас є купа `H` з коренем `A` (ключ 10). Ми хочемо вставити новий вузол `B` (ключ 5).

До вставки:

```
  H.root -> (A, 10)
             /  \
            ...
```

Після вставки B (використовуючи Meld):

Оскільки ключ B менший за ключ A, B стає новим коренем. `Meld` об’єднає `A` з правим піддеревом `B` (яке зараз `NULL`), а потім `B` обміняє своїх дітей.

```
  H.root -> (B, 5)
             / \
           NIL (A, 10)
               / \
              ...
```

Алгоритм:

```
Skew-Heap-Insert(H, x)
  // Ініціалізація нового вузла
```

```
  x.left_child = NULL
  x.right_child = NULL

  // Об'єднати H з новим вузлом (як окремою купою )
  H.root = Skew-Heap-Meld(H.root, x)
```

Псевдокод:

```
FUNCTION Skew-Heap-Insert(H, x):
  // Ініціалізація вузла x
  x.left_child = NIL
  x.right_child = NIL

  // Об'єднати H з новим вузлом x
  H.root = Skew-Heap-Meld(H.root, x)
```

Пошук мінімального елемента `Skew-Heap-Minimum(H)`

Ця операція повертає вузол з найменшим ключем у скошеній купі `H`. Завдяки властивості міні-купи, мінімальний елемент завжди є коренем дерева.

Опис: скошена купа зберігає посилання на кореневий вузол у полі `H.root`. Оскільки корінь завжди має найменший ключ серед усіх вузлів у купі, достатньо просто повернути значення цього поля. Ця операція займає константний час $O(1)$.

Візуалізація роботи:

Припустимо, у нас є купа `H` з коренем `B`.

```
  H.root -> (B, 5)  (мінімальний)
             / \
            ...
```

Алгоритм:

```
Skew-Heap-Minimum(H)
  Повернути H.root
```

Псевдокод:

```
FUNCTION Skew-Heap-Minimum(H):
  RETURN H.root
```

Вилучення мінімального елемента `Skew-Heap-Extract-Min(H)`

Ця операція видаляє вузол з найменшим ключем (корінь) зі скошеної купи `H` і повертає його. Це одна з основних операцій, яка також використовує `Meld`.

Опис: Вилучення мінімального елемента включає такі кроки:

1. Збереження кореня: Зберегти посилання на поточний кореневий вузол `z = H.root`.
2. Перевірка порожньої купи: Якщо `z` є `NULL`, купа порожня, повертаємо `NULL`.
3. Оновлення кореня: Новий корінь купи утворюється шляхом об'єднання лівого та правого дочірніх вузлів видаленого кореня `z`.
    - `H.root = Skew-Heap-Meld(z.left_child, z.right_child)`.
4. Очищення посилань: Встановити `z.left_child = NULL` та `z.right_child = NULL` для видаленого вузла `z`, щоб він був повністю від'єднаний від купи.

Візуалізація `Extract-Min`:

До `Extract-Min`:

```
H.root -> (A, 5)
           /   \
          (B, 10) (C, 15)
         /   \
        (D, 20) (E, 25)
```

Після `Extract-Min`:

– Видаляємо `A`.

– Об'єднуємо `B` та `C` за допомогою `Meld`. Припустимо, `B.key < C.key`, тому `B` стане коренем.

– `Meld` також обміняє дітей `B`.

```
H.root -> (B, 10)
           /   \
          (C, 15) (D, 20)  (D був лівим дітиною B, C був правим)
                  /  \
                 (E, 25) NIL
```

Алгоритм `Skew-Heap-Extract-Min(H)`:

```
Skew-Heap-Extract-Min(H)
  z = H.root
  Якщо z == NULL:
    Повернути NULL

  H.root = Skew-Heap-Meld(z.left_child, z.right_child) // Об'єднуємо
дочірні вузли видаленого кореня

  // Очистити посилання з видаленого вузла для коректного повернення
  z.left_child = NULL
  z.right_child = NULL

  Повернути z
```

Псевдокод `Skew-Heap-Extract-Min(H)`:

```
FUNCTION Skew-Heap-Extract-Min(H):
  z = H.root
  IF z IS NIL:
    RETURN NIL

  H.root = Skew-Heap-Meld(z.left_child, z.right_child)

  // Очистити посилання з видаленого вузла
  z.left_child = NIL
  z.right_child = NIL

  RETURN z
```

Об’єднання двох скошених куп `Skew-Heap-Meld(H1, H2)`

Ця операція об’єднує дві скошені купи **`H1`** та **`H2`** в одну нову купу. Вона рекурсивно об’єднує купи, підтримуючи властивість міні-купи та виконуючи обов’язковий обмін дочірніми вузлами.

Опис: Функція **`Meld`** приймає два корені дерев **`node1`** та **`node2`** (які можуть бути **`NULL`**) і повертає корінь нового об’єднаного дерева.

1. Базові випадки:
    - Якщо **`node1`** є **`NULL`**, повертаємо **`node2`**.
    - Якщо **`node2`** є **`NULL`**, повертаємо **`node1`**.

2. Визначення кореня: Порівняти ключі `node1.key` та `node2.key`. Вузол з меншим ключем стає коренем нового дерева. Нехай це буде `root_node`, а інший вузол — `other_node`.
    - Якщо `node1.key <= node2.key`: `root_node = node1`, `other_node = node2`.
    - Інакше: `root_node = node2`, `other_node = node1`.
3. Рекурсивне об'єднання: Праве піддерево `root_node` (тобто `root_node.right_child`) рекурсивно об'єднується з `other_node`. Результат цього об'єднання стає новим правим дочірнім вузлом `root_node`.
    - `root_node.right_child = Skew-Heap-Meld(root_node.right_child, other_node)`.
4. ОБОВ'ЯЗКОВИЙ ОБМІН ДОЧІРНІМИ ВУЗЛАМИ: Після рекурсивного виклику `Meld` і перед поверненням `root_node`, завжди обмінюємо місцями лівий та правий дочірні вузли `root_node`. Це є відмінною рисою скошеної купи.
    - `SWAP(root_node.left_child, root_node.right_child)`.

Візуалізація `Meld`:

Meld(H1, H2)

```
H1.root -> (A, 5)            H2.root -> (F, 6)
           /  \                        / \
         (B, 10) (C, 15)               (G, 12) (H, 18)
```

Крок 1: A.key (5) < F.key (6). Тому A - корінь, F - інший вузол.

Рекурсивно викликаємо `Meld(A.right_child (C), F)`.

Meld(C, F)

```
C (15)                  F (6)
 / \                      / \
NIL NIL                  (G, 12) (H, 18)
```

Крок 2: F.key (6) < C.key (15). Тому F - корінь, C - інший вузол.

Рекурсивно викликаємо `Meld(F.right_child (H), C)`.

Meld(H, C)

```
H (18)                  C (15)
 / \                      / \
NIL NIL                 NIL NIL
```

Крок 3: C.key (15) < H.key (18). Тому C - корінь, H - інший вузол.

Рекурсивно викликаємо `Meld(C.right_child (NIL), H)`. Результат `H`.

Повернення з рекурсії:

- `Meld(C.right_child (NIL), H)` повертає `H`.
    - Обмін у C (але C не є коренем): Ні. Це відбувається тільки для `root_node` під час повернення.
- Результат `Meld(H, C)` стає `F.right_child`. (Припустимо, що `Meld(C, H)` поверне `C` і `H` стане її правим дочірнім, потім `C` обміняє дітей, і `H` стане лівим дочірнім, а `NIL` - правим).
    - Обмін у F: `F.left_child` (G) та `F.right_child` (нове дерево з коренем C) міняються місцями.
    - Повертаємо `F`.
- Результат `Meld(A.right_child, F)` стає `A.right_child`.
    - Обмін у A: `A.left_child` (B) та `A.right_child` (нове дерево з коренем F) міняються місцями.
    - Повертаємо `A`.

Остаточний результат Meld(H1, H2):

```
H.root -> (A, 5)
            /   \
          (F, 6) (B, 10)
         /   \
        (C, 15) (G, 12)
       /   \
      NIL   (H, 18)
```

Алгоритм `Skew-Heap-Meld(H1_node, H2_node)`:

```
Skew-Heap-Meld(H1_node, H2_node)
  // Базові випадки: якщо одна з куп порожня
```

```
  Якщо H1_node == NULL:
    Повернути H2_node
  Якщо H2_node == NULL:
    Повернути H1_node

  // Визначити корінь об'єднаної купи (вузол з меншим ключем)
  Якщо H1_node.key > H2_node.key:
    SWAP(H1_node, H2_node) // Забезпечуємо, що H1_node завжди має мен
ший ключ

  // Рекурсивно об'єднати праве піддерево кореня з іншою купою
  H1_node.right_child = Skew-Heap-Meld(H1_node.right_child, H2_node)

  // ОБОВ'ЯЗКОВИЙ ОБМІН ДОЧІРНІМИ ВУЗЛАМИ:
  // Обміняти лівий та правий дочірні вузли H1_node
  SWAP(H1_node.left_child, H1_node.right_child)

  Повернути H1_node
```

Псевдокод `Skew-Heap-Meld(H1_node, H2_node)`:

```
FUNCTION Skew-Heap-Meld(H1_node, H2_node):
  // Базові випадки
  IF H1_node IS NIL:
    RETURN H2_node
  IF H2_node IS NIL:
    RETURN H1_node

  // Переконатися, що H1_node має менший ключ
  IF H1_node.key > H2_node.key:
    SWAP(H1_node, H2_node)

  // Рекурсивно об'єднати праве піддерево H1_node з H2_node
  H1_node.right_child = Skew-Heap-Meld(H1_node.right_child, H2_node)

  // ОБОВ'ЯЗКОВИЙ ОБМІН ДОЧІРНІМИ ВУЗЛАМИ
  SWAP(H1_node.left_child, H1_node.right_child)

  RETURN H1_node
```

Зменшення ключа вузла `Skew-Heap-Decrease-Key(H, x, k)`

Ця операція зменшує ключ вузла `x` до нового значення `k`. Важливо, що `k` повинно бути меншим за поточний ключ `x.key`. Якщо після зменшення

ключа властивість міні-купи порушується (тобто `x.key < x.parent.key`), то вузол `x` необхідно від'єднати від його батька і об'єднати з коренем купи.

Опис:

1. Перевірка вхідних даних: Переконатися, що новий ключ `k` дійсно менший за поточний ключ `x.key`. Якщо ні, це є помилкою.
2. Оновлення ключа: Встановити `x.key = k`.
3. Перевірка порушення властивості купи: Якщо `x` не є коренем купи (`x` має батька `p`) і `x.key < p.key`:
    - Від'єднання `x`: Вузол `x` необхідно тимчасово від'єднати від його батька `p`. Це означає, що посилання `p.left_child` або `p.right_child`, яке вказувало на `x`, має бути встановлено на `NULL`.
    - Перебалансування батьківського піддерева: Після від'єднання `x`, батьківське дерево, коренем якого був `p`, може потребувати ребалансування. Це робиться шляхом об'єднання його (нового) лівого та правого піддерев.
    - Об'єднання з коренем: Від'єднаний вузол `x` (який тепер є окремою Скошеною Купою з новим ключем) об'єднується з поточною Купою `H.root` за допомогою `Meld`.

Як і у випадку з лівосторонніми купами, для ефективної реалізації `Decrease-Key` "на місці" вузли скошеної купи повинні мати посилання на своїх батьків (`parent`). Без них реалізація стає значно складнішою або вимагає операції "видалення-та-вставки".

Алгоритм `Skew-Heap-Decrease-Key(H, x, k)`:

```
Skew-Heap-Decrease-Key(H, x, k)
  Якщо k > x.key:
    Помилка "Новий ключ більший за поточний"

  // Якщо x є коренем, або новий ключ не порушує властивість купи, пр
осто оновлюємо ключ
  Якщо x == H.root:
    x.key = k
    Повернути

  parent = x.parent // Припускаємо наявність посилання на батька
```

```
  // Якщо новий ключ все ще більший або дорівнює ключу батька, просто оновлюємо ключ
  // і властивість купи не порушується
  Якщо k >= parent.key:
    x.key = k
    Повернути

  // Якщо властивість купи порушується (новий ключ x менший за батьківський)
  x.key = k // Оновлюємо ключ

  // Від'єднуємо вузол x від його батька
  Якщо parent.left_child == x:
    parent.left_child = NULL
  Інакше:
    parent.right_child = NULL

  // Тепер, parent_node.left_child або parent_node.right_child став NULL.
  // Батьківське дерево може потребувати перебалансування.
  // Для цього ми об'єднуємо залишки його піддерев.
  parent_replacement_subtree = Skew-Heap-Meld(parent.left_child, parent.right_child)

  // Оновлюємо посилання батька на його нове піддерево (якщо воно існує)
  // Якщо parent_replacement_subtree не NULL, йому потрібен parent.
  // Але тут ми просто перевстановлюємо children parent_node.

  // Перепризначити дочірні вузли parent_node
  parent.left_child = parent_replacement_subtree
  parent.right_child = NULL // Забезпечуємо, що parent.right_child стане NULL, готовий до обміну

  // Потім, під час наступної операції Meld, що зачіпає parent_node,
  // він також пройде обмін дітей.

  // Зробити x окремим деревом.
  x.parent = NULL // x більше не має батька
  x.left_child = NULL
  x.right_child = NULL

  // Об'єднати x з поточною купою
  H.root = Skew-Heap-Meld(H.root, x)
```

Псевдокод Skew-Heap-Decrease-Key(H, x, k):

```
FUNCTION Skew-Heap-Decrease-Key(H, x, k):
  IF k > x.key:
    ERROR "New key is greater than current key."

  // Якщо x вже корінь, або новий ключ не порушує властивість купи
  IF x IS H.root OR k >= x.parent.key: // Припускаємо x.parent існує якщо x не корінь
    x.key = k
    RETURN

  // Властивість купи порушена
  x.key = k

  parent_node = x.parent // Припускаємо наявність x.parent

  // Від'єднуємо x від його батька
  IF parent_node.left_child IS x:
    parent_node.left_child = NIL
  ELSE:
    parent_node.right_child = NIL

  // Перебалансовуємо батьківське дерево після від'єднання дитини.
  // Це відбувається шляхом об'єднання його поточних лівого та правого піддерев.
  // Це важливо для підтримки амортизованої продуктивності.
  // Якщо parent_node.left_child або parent_node.right_child стали NIL після від'єднання x,
  // то Meld(parent_node.left_child, parent_node.right_child) коректно обробить це.

  // Створюємо тимчасовий корінь з об'єднаних піддерев батьківського вузла.
  temp_parent_child = Skew-Heap-Meld(parent_node.left_child, parent_node.right_child)

  // Призначаємо його лівим дочірнім для parent_node
  parent_node.left_child = temp_parent_child
  parent_node.right_child = NIL // Правий дочірній стає NIL для майбутніх обмінів

  // Встановлюємо батька для нового тимчасового дочірнього вузла
  IF temp_parent_child IS NOT NIL:
    temp_parent_child.parent = parent_node
```

```
// Від'єднуємо x повністю від батьківського посилання
x.parent = NIL
x.left_child = NIL
x.right_child = NIL

// Об'єднуємо модифікований вузол x з коренем купи.
H.root = Skew-Heap-Meld(H.root, x)
```

Видалення вузла `Skew-Heap-Delete(H, x)`

Ця операція видаляє довільний вузол `x` зі скошеної купи `H`. Це може бути виконано шляхом зменшення ключа `x` до негативної нескінченності, а потім викликом `Extract-Min`.

Опис:

1. Зменшення ключа: Викликати `Skew-Heap-Decrease-Key(H, x, -infinity)` (або дуже мале число, наприклад, `MIN_INT`). Це гарантує, що `x` стане мінімальним елементом купи.
2. Вилучення мінімуму: Викликати `Skew-Heap-Extract-Min(H)`. Це видалить `x` (який тепер є мінімальним елементом) з купи.

Візуалізація роботи:

До `Delete(H, G)` (G має ключ 12):

```
H.root -> (A, 5)
           /   \
         (B, 10) (C, 15)
        /   \
      (G, 12) (E, 25)
```

Крок 1: `Decrease-Key(H, G, -infinity)`

`G` стає кореневим вузлом (або приєднується до кореня `H`), і `H.root` вказує на `G`.

```
H.root -> (G, -infinity)
           /   \
         (A, 5) (решта вузлів A)
```

Крок 2: `Extract-Min(H)`

`G` (який тепер є мінімальним) видаляється. Дочірні вузли `G` обробляються за допомогою `Meld`, щоб сформувати новий корінь купи.

Після `Delete`:

```
  H.root -> (A, 5) (залежить від початкового H.root та G.children)
              /   \
            (B, 10) (C, 15)
           /   \
         NIL   (E, 25)
```

Алгоритм:

```
Skew-Heap-Delete(H, x)
  Skew-Heap-Decrease-Key(H, x, -infinity) // Зменшуємо ключ до наймен
шого можливого значення
  Skew-Heap-Extract-Min(H) // Видаляємо теперішній мінімальний елемен
т (який є x)
```

Псевдокод:

```
FUNCTION Skew-Heap-Delete(H, x):
  // Припускаємо, що є константа NEGATIVE_INFINITY
  Skew-Heap-Decrease-Key(H, x, NEGATIVE_INFINITY)
  Skew-Heap-Extract-Min(H)
```

Обходи Дерева

Оскільки скошена купа є бінарним деревом, до неї застосовуються стандартні алгоритми обходу бінарних дерев: прямий обхід (Pre-order), центрований обхід (In-order), зворотний обхід (Post-order) та обхід у ширину (BFS).

Структура вузла для обходу

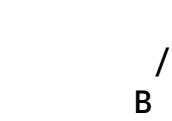
```
Структура Вузла:
  key: int
  left_child: Node*
  right_child: Node*
  // parent: Node* (опціонально, але корисно для Decrease-Key/Delete)
```

3. Обхід у глибину (Depth-First Search - DFS)

1.1. Прямий обхід

Відвідати кореневий вузол, потім рекурсивно відвідати ліве піддерево, потім праве піддерево.

Візуалізація роботи:




```
Порядок обходу: A -> B -> C
```

Алгоритм:

```
Pre-order-Traversal(node)
  Якщо node == NULL:
    Повернути
  Відвідати(node) // Наприклад, вивести node.key

  Pre-order-Traversal(node.left_child)
  Pre-order-Traversal(node.right_child)
```

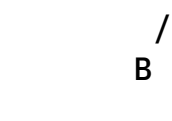
Псевдокод:

```
PROCEDURE Pre-order-Traversal(node):
  IF node IS NIL:
    RETURN

  VISIT(node) // Наприклад, PRINT node.key

  Pre-order-Traversal(node.left_child)
  Pre-order-Traversal(node.right_child)
```

1.2. Центрований обхід

Рекурсивно відвідати ліве піддерево, потім відвідати кореневий вузол, потім рекурсивно відвідати праве піддерево.

Візуалізація роботи:




```
Порядок обходу: B -> A -> C
```

Алгоритм:

```
In-order-Traversal(node)
  Якщо node == NULL:
```

```
    Повернути

  In-order-Traversal(node.left_child)
  Відвідати(node) // Наприклад, вивести node.key
  In-order-Traversal(node.right_child)
```

Псевдокод:

```
PROCEDURE In-order-Traversal(node):
  IF node IS NIL:
    RETURN

  In-order-Traversal(node.left_child)
  VISIT(node) // Наприклад, PRINT node.key
  In-order-Traversal(node.right_child)
```

1.3. Зворотний обхід

Рекурсивно відвідати ліве піддерево, потім праве піддерево, а потім відвідати кореневий вузол.

Візуалізація роботи:

```
      A
     / \
    B   C

Порядок обходу: B -> C -> A
```

Алгоритм:

```
Post-order-Traversal(node)
  Якщо node == NULL:
    Повернути

  Post-order-Traversal(node.left_child)
  Post-order-Traversal(node.right_child)
  Відвідати(node) // Наприклад, вивести node.key
```

Псевдокод:

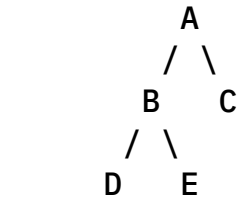
```
PROCEDURE Post-order-Traversal(node):
  IF node IS NIL:
    RETURN

  Post-order-Traversal(node.left_child)
```

```
  Post-order-Traversal(node.right_child)
  VISIT(node) // Наприклад, PRINT node.key
```

Обхід у ширину

Відвідати вузли рівень за рівнем, починаючи від кореня. Використовує чергу для відстеження вузлів, які потрібно відвідати.

Візуалізація роботи:




```
Порядок обходу: A -> B -> C -> D -> E
```

Алгоритм:

```
BFS-Traversal(root_node)
  Якщо root_node == NULL:
    Повернути

  queue = Нова_Черга()
  queue.Додати(root_node)

  Поки queue не порожня:
    current_node = queue.Видалити()
    Відвідати(current_node) // Наприклад, вивести current_node.key

    Якщо current_node.left_child != NULL:
      queue.Додати(current_node.left_child)
    Якщо current_node.right_child != NULL:
      queue.Додати(current_node.right_child)
```

Псевдокод:

```
PROCEDURE BFS-Traversal(root_node):
  IF root_node IS NIL:
    RETURN

  queue = NEW QUEUE()
  queue.ENQUEUE(root_node)
```

```
  WHILE NOT queue.IS_EMPTY():
    current_node = queue.DEQUEUE()
    VISIT(current_node) // Наприклад, PRINT current_node.key

    IF current_node.left_child IS NOT NIL:
      queue.ENQUEUE(current_node.left_child)
    IF current_node.right_child IS NOT NIL:
      queue.ENQUEUE(current_node.right_child)
```

Обхід всієї скошеної купи

Оскільки скошена купа є одним деревом, для обходу всієї купи достатньо викликати відповідну функцію обходу для її кореня:

```
Skew-Heap-Full-Traversal(H)
  Pre-order-Traversal(H.root) // Або In-order-Traversal, Post-order-T
raversal, BFS-Traversal
```

2. Область застосування

Скошені купи є відмінним вибором для сценаріїв, де операція об’єднання (meld) двох пріоритетних черг є частою і критично важливою, а також коли важлива простота реалізації та компактність структури. Їхня ефективність об’єднання, яка поширюється на інші операції, робить їх конкурентоспроможними в багатьох алгоритмах.

Основними областями застосування скошених куп є:

- Алгоритми на графах: Подібно до лівосторонніх куп, скошені купи можуть бути використані в алгоритмах Дейкстри та Прима, де вони забезпечують таку ж асимптотичну продуктивність, як і лівосторонні купи. Їхня простота робить їх привабливими.
- Симуляції дискретних подій: У системах, де потрібно керувати чергою подій, і де може виникнути потреба об’єднати списки подій з різних джерел.
- Алгоритми, що вимагають частих злиттів: Будь-яка задача, де злиття пріоритетних черг є основним вузьким місцем.
- Дисперсні структури даних: Де важлива мінімальна кількість додаткової інформації, що зберігається у вузлах.

– Деякі алгоритми для обробки паралельних даних: Можливість ефективно об’єднувати черги з різних потоків або процесів.

3. Приклади застосування

Давайте розглянемо конкретні приклади алгоритмів, де скошені купи можуть бути застосовані.

3.1. Алгоритм Дейкстри для пошуку найкоротшого шляху

Алгоритм Дейкстри знаходить найкоротші шляхи від одного джерела до всіх інших вершин у графі з невід’ємними вагами ребер. Ефективність цього алгоритму сильно залежить від реалізації пріоритетної черги.

Подібно до лівосторонніх куп, для `Decrease-Key` у скошеній купі можна використовувати стратегію “видалення-та-вставки” (`Delete` + `Insert`) або більш складний підхід з використанням посилань на батьків. В обох випадках, завдяки амортизованій логарифмічній складності операцій, скошені купи забезпечують ефективність, подібну до бінарних куп ($O(E \log V)$), але з перевагою простоти реалізації та ефективності `Meld`.

– Стратегія “видалення-та-вставки”: Коли знайдено коротший шлях до вершини v, вузол v видаляється з купи, а потім вставляється назад з новим, меншим ключем. Це дві $O(\log N)$ операції.

– Стратегія з посиланнями на батьків: Якщо вузли мають посилання на батьків, вузол v від’єднується від свого батька, а потім об’єднується з коренем купи. Цей підхід є більш “чистим” для `Decrease-Key`.

3.2. Алгоритм Прима для побудови мінімального остовного дерева

Алгоритм Прима будує мінімальне остовне дерево (MST) у зв’язаному, зваженому, неорієнтованому графі. Він також активно використовує пріоритетну чергу.

Аналогічно алгоритму Дейкстри, скошені купи можуть ефективно використовуватися в алгоритмі Прима для керування пріоритетами вершин. Операція `Decrease-Key` буде відігравати ключову роль.

3.3. Об’єднання багатьох пріоритетних черг

У сценаріях, де необхідно регулярно об’єднувати велику кількість невеликих пріоритетних черг в одну, або коли дані надходять з різних джерел, і їх потрібно консолідувати.

Завдяки своїй $O(\log N)$ амортизованій складності для операції `Meld` у найгіршому випадку, скошені купи є дуже ефективним вибором. Це може бути використано в:

- Алгоритмах злиття даних: Коли результати кількох паралельних процесів, які генерують пріоритетні дані, потрібно об'єднати.
- Системах управління ресурсами: Де завдання з різними пріоритетами з різних джерел потрібно об'єднати в єдину чергу для планування.

Переваги та Недоліки

Скошені купи є цікавим компромісом, оскільки вони не підтримують явних властивостей збалансованості, але все одно досягають хорошої продуктивності завдяки постійним обмінам дочірніми вузлами.

Переваги

1. Простота реалізації: Це, мабуть, одна з найбільших переваг. скошені купи надзвичайно прості в реалізації порівняно з купами Фібоначчі, біноміальними купами або навіть лівосторонніми купами (які вимагають поля `npl` та його підтримки). Їхня простота походить від відсутності складних властивостей збалансованості, окрім базової властивості купи.
2. Ефективна операція об'єднання (`Meld`): Операція `Meld` виконується за $O(\log n)$ амортизованого часу в найгіршому випадку. Це дуже ефективно і робить їх привабливими для алгоритмів, що часто потребують злиття.
3. Хороша амортизована продуктивність: Всі основні операції (`Insert`, `Extract-Min`, `Meld`, `Decrease-Key`, `Delete`) мають амортизовану часову складність $O(\log n)$. Це означає, що, хоча окремі операції можуть бути дорожчими, послідовність операцій буде ефективною в середньому.
4. Компактність вузла: Вузол скошеної купи потребує лише ключ та посилання на двох дочірніх вузлів. Немає додаткових полів, таких як `npl` або `marked`, що зменшує накладні витрати на пам'ять для кожного вузла.

Недоліки

1. Амортизована, а не гарантована часова складність: Основний недолік полягає в тому, що продуктивність є амортизованою. Це означає, що окремі операції можуть займати $O(N)$ часу у найгіршому випадку (наприклад, якщо купа сильно “скошена” після серії операцій `Meld`), хоча серія операцій у середньому буде логарифмічною. Це може бути проблемою для систем реального часу, де кожна операція повинна завершитися протягом певного часового вікна.

2. Відсутність гарантованої збалансованості: Оскільки скошена купа не має явних правил збалансованості (крім обов’язкового обміну дітей), її форма може бути дуже “скошеною” або “витягнутою”, що може призвести до високих окремих операційних витрат.

3. Необхідність посилань на батьків для `Decrease-Key` / `Delete`: Ефективна реалізація `Decrease-Key` та `Delete` вимагає, щоб вузли мали посилання на своїх батьків. Якщо таких посилань немає, ці операції стають значно дорожчими (наприклад, через пошук вузла).

Скошені купи є елегантним та практичним вибором для пріоритетних черг, особливо коли важлива простота реалізації та ефективне об’єднання. Вони пропонують хорошу амортизовану продуктивність без складності, властивої купам Фібоначчі, хоча їхня амортизована природа може бути обмеженням у деяких строгих сценаріях.

Контрольні запитання

1. У чому полягає ключова відмінність скошеної купи від лівосторонньої купи?

2. Які поля даних містить кожен вузол скошеної купи?

3. Яка операція є фундаментальною для всіх інших операцій у скошеній купі?

4. Що відбувається з дочірніми вузлами кореня після кожного рекурсивного виклику `Meld` у скошеній купі?

5. Яка асимптотична часова складність операції `Insert` у скошеній купі?

6. Чому скошена купа вважається “скошеною”?

7. Наведіть приклад алгоритму, де скошена купа може бути ефективно використана.
8. Які є основні переваги скошеної купи порівняно з купою Фібоначчі?
9. Який основний недолік скошеної купи для систем реального часу?
10. Поясніть, як `Delete` операція зазвичай реалізується у скошеній купі.

Методичні Вказівки до лабораторної роботи

1. Назва

Лабораторна робота: “Реалізація та Аналіз скошеної купи”.

2. Мета

Метою даної лабораторної роботи є закріплення теоретичних знань про структуру та операції скошеної купи, а також набуття практичних навичок у її реалізації та тестуванні. Робота спрямована на глибоке розуміння принципу “безумовного обміну” та амортизованої часової складності без явних властивостей збалансованості.

3. Завдання
4. Реалізувати структуру даних “скошена купа” з усіма основними операціями:
    - `Make-Skew-Heap()`
    - `Skew-Heap-Insert(H, x)`
    - `Skew-Heap-Minimum(H)`
    - `Skew-Heap-Extract-Min(H)`
    - `Skew-Heap-Meld(H1, H2)`
    - `Skew-Heap-Decrease-Key(H, x, k)` (реалізувати з батьківськими посиланнями)
    - `Skew-Heap-Delete(H, x)`
5. Розробити тестовий набір для перевірки коректності реалізації кожної операції.

6. Продемонструвати роботу скошеної купи на прикладі задачі, що вимагає ефективного об’єднання черг.

7. Опис

Скошена купа — це бінарне дерево, що підтримує властивості міні-купи. Кожен вузол містить ключ, посилання на лівого та правого дочірніх вузлів, а також посилання на батька (`parent`) для ефективної реалізації `Decrease-Key` та `Delete`. Ключовою особливістю є безумовний обмін лівого та правого дочірніх вузлів кореня після кожного об’єднання (`Meld`). Реалізація вимагає рекурсивного підходу до `Meld` та уважного керування посиланнями.

5. Порядок Виконання

   a) Визначення структури вузла: Створіть клас або структуру `SkewHeapNode` (або аналогічну назву) з полями `key`, `left_child`, `right_child`, `parent`.
   b) Визначення структури купи: Створіть клас або структуру `SkewHeap` (або аналогічну назву) з полем `root` (або `H.root`).
   c) Реалізація `Make-Skew-Heap()`: Ініціалізуйте `root` як `NULL`.
   d) Реалізація `Skew-Heap-Meld(H1_node, H2_node)`: Це центральна функція. Реалізуйте її рекурсивно, враховуючи базові випадки, порівняння ключів, рекурсивне об’єднання правого піддерева та обов’язковий обмін лівого та правого дочірніх вузлів кореня перед поверненням. Оновіть `parent` посилання для нового дочірнього вузла.
   e) Реалізація `Skew-Heap-Insert(H, x)`: Використайте `Meld` для об’єднання купи з новим вузлом.
   f) Реалізація `Skew-Heap-Minimum(H)`: Поверніть `H.root`.
   g) Реалізація `Skew-Heap-Extract-Min(H)`: Використайте `Meld` для об’єднання лівого та правого дочірніх вузлів видаленого кореня. Не забудьте очистити `parent` посилання для цих вузлів.
   h) Реалізація `Skew-Heap-Decrease-Key(H, x, k)`: Реалізуйте її з використанням посилань на батьків. Якщо властивість купи порушена: від’єднайте `x` від його батька, зробіть `x` окремим деревом (з `parent=NULL`, `left_child=NULL`, `right_child=NULL`), перебалансуйте батьківське дерево за допомогою `Meld` його дочірніх вузлів, а потім об’єднайте `x` з `H.root` за допомогою `Meld`.

i) Реалізація `Skew-Heap-Delete(H, x)`: Реалізуйте її шляхом виклику `Decrease-Key(H, x, -infinity)` та `Extract-Min(H)`. Це вимагатиме, щоб вузол `x` мав посилання на батька.

6. Тестування:

   - Створіть декілька тестових випадків для кожної операції, що охоплюють різні сценарії (порожня купа, купа з одним елементом, складні сценарії об'єднання, вставки, зменшення ключа).
   - Використайте функцію обходу дерева (наприклад, прямий обхід) для візуалізації структури купи після ключових операцій. Зверніть увагу на "скошеність" дерева.

7. Демонстрація застосування: Реалізуйте сценарій, де кілька пріоритетних черг (наприклад, від різних "джерел") об'єднуються в одну головну чергу. Наприклад, симуляція подій з можливістю об'єднання черг подій.

8. Результат

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де реалізована структура даних "скошена купа" з усіма описаними операціями та допоміжними функціями, а також приклади тестування кожної операції. Код повинен бути чистим, добре коментованим та відповідати принципам об'єктно-орієнтованого програмування.
2. Продемонструвати роботу додатку: показати виконання ключових операцій (`Insert`, `Extract-Min`, `Meld`, `Decrease-Key`, `Delete`) на конкретних прикладах, звертаючи увагу на структуру дерева та обміни дочірніми вузлами після `Meld`. Візуалізація структури купи (наприклад, текстова схема, що показує зв'язки) після критичних операцій є обов'язковою.
3. Бути готовим відповісти на питання:

   - Поясніть, чому скошені купи не потребують додаткових властивостей збалансованості (як `npl`) для досягнення амортизованої ефективності.

- Опишіть, як безумовний обмін дочірніми вузлами у `Meld` допомагає підтримувати амортизовану продуктивність.
- Які є компроміси між простотою реалізації скошеної купи та її амортизованою природою?
- Поясніть, як реалізовано `Decrease-Key` у вашій програмі, і які переваги надає наявність `parent` посилань.
- У яких випадках скошена купа буде кращим вибором, ніж Лівостороння Купа, а коли – гіршим?

Тести

1. Яке поле *відсутнє* у вузлі скошеної купи, але присутнє у вузлі Лівосторонньої Купи?
    a) `key`
    b) `left_child`
    c) `parent`
    d) `npl`
2. Яка операція в скошеній купі є найефективнішою і фундаментальною, на якій базуються інші?
    a) `Insert`
    b) `Extract-Min`
    c) `Meld`
    d) `Decrease-Key`
3. Що відбувається з дочірніми вузлами кореня *після кожного* рекурсивного виклику `Meld` у скошеній купі?
    a) Вони залишаються на своїх місцях.
    b) Вони сортуються за ключем.
    c) Вони міняються місцями.
    d) Один з них видаляється.

4. Яка асимптотична часова складність операції **`Meld`** у скошеній купі?

    a) $O(1)$

    b) $O(\log n)$ (амортизовано)

    c) $O(n)$

    d) $O(n \log n)$

5. Який основний недолік скошеної купи для застосунків, що вимагають гарантованого часу виконання кожної окремої операції?

    a) Високі накладні витрати на пам’ять.

    b) Складність реалізації.

    c) Продуктивність є амортизованою, а не гарантованою в найгіршому випадку.

    d) Відсутність операції **`Delete`**.

Відповіді: 1. – d), 2. – c), 3. – c), 4. – b), 5. - c)

Тести підвищеної складності

1. Розгляньте послідовність з M операцій **`Insert`** та K операцій **`Extract-Min`** на порожній скошеній купі. Яка амортизована часова складність цієї послідовності?

    a) $O(M + K)$

    b) $O((M + K)\log(M + K))$

    c) $O(M \log M + K)$

    d) $O(M \cdot K \log N)$

2. Чому скошена купа не вимагає підтримки таких властивостей, як **`npl`** (як у Лівосторонній Купі) або біноміальних дерев (як у Біноміальній Купі)?

    a) Вона працює тільки з дуже малими даними.

    b) Її амортизована продуктивність забезпечується випадковим обміном дочірніми вузлами.

    c) Вона є просто бінарним пошуковим деревом.

d) Вона не підтримує операції об’єднання.

3. Якщо `Skew-Heap-Meld(H1_node, H2_node)` викликається з `H1_node.key > H2_node.key`, що відбудеться на початку функції?

   a) `H1_node` буде відкинуто.

   b) `H2_node` буде приєднано до `H1_node` як лівий дочірній.

   c) `H1_node` та `H2_node` будуть обміняні місцями, щоб `H1_node` мав менший ключ.

   d) Виникне помилка.

4. Яка з наступних структур даних є найбільш простою в реалізації серед тих, що підтримують ефективну операцію `Meld`?

   a) Купа Фібоначчі

   b) Біноміальна купа

   c) Лівостороння купа

   d) скошена купа

5. Якщо вузол `x` є коренем скошеної купи. Що поверне `Skew-Heap-Minimum(H)`?

   a) `x.key`

   b) `H.root.left_child`

   c) `x`

   d) `NULL`

6. Який тип алгоритмів на графах найбільше виграє від ефективної операції `Meld` у скошеній купі?

   a) Алгоритми пошуку в глибину (DFS)

   b) Алгоритми, які часто об’єднують множини вершин (наприклад, деякі MST алгоритми)

   c) Алгоритми, що вимагають лише швидкого пошуку мінімуму

   d) Алгоритми, що працюють з негативними вагами ребер

7. Яка максимальна висота може бути у скошеної купи з N вузлами у найгіршому випадку (не амортизовано)?

    a) $O(\log N)$

    b) $O(N)$

    c) $O(\sqrt{N})$

    d) $O(N \log N)$

8. Який елемент вузла `x` потрібно мати для ефективної реалізації `Decrease-Key` у скошеній купі?

    a) `x.npl`

    b) `x.parent`

    c) `x.degree`

    d) `x.child`

9. Яке твердження про амортизовану часову складність у скошеній купі є вірним?

    a) Кожна операція завжди завершується за логарифмічний час.

    b) Окремі операції можуть бути повільними, але серія операцій у середньому ефективна.

    c) Вона застосовується лише до операції `Insert`.

    d) Це гарантія найгіршого випадку для будь-якої операції.

10. Порівняйте скошену купу з лівосторонньою упою . Яка з них має більші накладні витрати на пам’ять на вузол і чому?

    a) Скошена купа, бо вона має більше посилань.

    b) Лівостороння купа, бо вона має додаткове поле `npl`.

    c) Вони мають однакові накладні витрати.

    d) Жодна, бо вони обидві дуже компактні.

Відповіді на тести підвищеної складності

1. b) Кожна операція `Insert` та `Extract-Min` у скошеній купі має амортизовану часову складність $O(\log N)$, де N — поточна кількість елементів. Загалом, для M вставок та K вилучень, загальна амортизована складність буде $(M + K) \times O(\log(M + K))$.

2. b) Постійний, безумовний обмін дочірніми вузлами у `Meld` ефективно "перемішує" структуру купи, запобігаючи її дегенерації до найгіршого випадку на довгій послідовності операцій.

3. c) `Meld` завжди починається з того, що гарантує, що перший аргумент (який стане `root_node`) має менший або рівний ключ порівняно з другим аргументом.

4. d) Завдяки відсутності додаткових полів та відносно простим правилам обміну, скошені купи є одними з найпростіших у реалізації серед ефективних merge-based куп.

5. c) Операція `Skew-Heap-Minimum(H)` повертає посилання на кореневий вузол купи, який за визначенням є вузлом `x` у цьому випадку.

6. b) Сильною стороною Скошених Куп є їхня ефективна операція `Meld`, що робить їх ідеальними для алгоритмів, які вимагають об'єднання пріоритетних черг.

7. b) У найгіршому (не амортизованому) випадку, скошена купа може дегенерувати до ланцюга (як витягнутий зв'язаний список), де висота дорівнює кількості вузлів N.

8. b) Для ефективної `Decrease-Key` (та `Delete`), необхідно мати можливість від'єднати вузол від його батька, що вимагає посилання `parent`.

9. b) Це є сутністю амортизованого аналізу. "Дорогі" операції (як ті, що викликають багато обмінів) рідкісні і "оплачуються" багатьма "дешевими" операціями.

10. b) Кожен вузол Лівосторонньої Купи зберігає додаткове цілочисельне поле `npl` для підтримки властивості лівосторонності, тоді як скошена купа не має таких додаткових полів.

Завдання для самостійної роботи

1. Порівняння “на практиці”: Якщо ви реалізували купи Фібоначчі, подвійна та лівосторонні купи в попередніх лабораторних роботах, проведіть емпіричне порівняння продуктивності всіх чотирьох типів куп (включно зі скошеною купою) на одному й тому ж наборі тестових даних для різних операцій. Зробіть висновки щодо їхньої практичної ефективності.

2. Аналіз амортизованої складності: Детально вивчіть, як саме доводиться амортизована часова складність для скошеної купи. Зосередьтеся на методі потенціалів та як він застосовується для оцінки вартості операцій.

3. Вплив обмінів: Створіть візуалізацію або покрокову демонстрацію, яка показує, як безумовний обмін дочірніми вузлами в `Meld` допомагає “перемішувати” дерево і запобігати його постійному викривленню в один бік.

4. Оптимізація `Decrease-Key` без `parent` посилань: Дослідіть можливі стратегії реалізації `Decrease-Key` для скошеної купи, якщо вузли не зберігають посилання на батьків. Наскільки це ускладнює реалізацію та впливає на ефективність?

5. Застосування у маршрутизації мереж: Дослідіть, як пріоритетні черги, зокрема скошені купи, можуть бути використані в алгоритмах маршрутизації в комп’ютерних мережах (наприклад, для керування таблицями маршрутизації або пріоритезації пакетів).

6. Генерація “найгіршого випадку”: Чи можете ви придумати послідовність операцій для скошеної купи, яка призведе до $O(N)$ часу для *окремої* операції? Якщо так, продемонструйте це.

7. Компактна реалізація: Розгляньте, як можна оптимізувати пам’ять для скошеної купи, наприклад, використовуючи масив для зберігання вузлів або інші техніки.

8. Симуляція динамічних подій: Реалізуйте симуляцію системи, де події генеруються динамічно, мають пріоритети, і можуть бути об’єднані або видалені. Використайте скошену купу як основу для черги подій.

9. Порівняння з бінарною купою : Проведіть експерименти, порівнюючи скошену купу зі звичайною бінарною купою для різних сценаріїв використання (наприклад, тільки `Insert`/`Extract-Min` проти сценаріїв з частими `Meld`). Коли скошена купа виграє?

10. Розширення функціональності: Додайте до вашої реалізації скошеної купи функцію `Heap-Merge-Sort` (сортування злиттям на основі купи), яка використовує операцію `Meld` для ефективного сортування елементів.

Частина III: Алгоритмічні парадигми

Розділ 6: Класичні парадигми проєктування

1. Розділяй і володарюй

Алгоритм множення Карацуби

1. Теоретичні відомості

1.1. Постановка задачі: проблема множення великих чисел

Здавалося б, що може бути простішим за множення двох чисел? Комп'ютери роблять це мільйони разів на секунду. Проте стандартні типи даних, як-от `int` чи `long long`, мають обмежений розмір. Що робити, якщо нам потрібно перемножити два числа, кожне з яких має, наприклад, тисячу знаків? Такі задачі є типовими для криптографії, наукових обчислень високої точності та комп'ютерної алгебри.

Для роботи з такими числами їх зазвичай представляють у вигляді масиву цифр або рядка. Давайте пригадаємо «шкільний» метод множення у стовпчик. Якщо ми множимо два n-значні числа, ми, по суті, множимо кожну цифру першого числа на кожну цифру другого. Це призводить до $n \times n = n^2$ одноцифрових множень. Отже, обчислювальна складність такого підходу — $O(n^2)$.

Протягом десятиліть вважалося, що ця складність є теоретично найкращою. Здавалося очевидним, що для отримання результату потрібно взаємодіяти

кожній цифрі з кожною. Але, як це часто буває в науці, очевидне не завжди є істинним.

1.2. Застосування «Розділяй і володарюй»: перша спроба

Як ми вже знаємо з попереднього розділу, парадигма «Розділяй і володарюй» може допомогти розв'язати складну задачу, розбивши її на менші частини. Спробуємо застосувати цей підхід.

Крок 1: Розділяй. Візьмемо два *n*-значні числа, X та Y. Розділимо кожне з них на дві половини, «старшу» та «молодшу», кожна розміром приблизно *n/2* цифр. Алгебраїчно це можна записати так:

- $X = a \cdot 10^{n/2} + b$
- $Y = c \cdot 10^{n/2} + d$

Пояснення позначень:

- a, c: старші половини чисел X та Y.
- b, d: молодші половини чисел X та Y.
- $10^{n/2}$: це «зсув», який ставить старшу половину на своє місце. Наприклад, для числа **5678** (де n=4, n/2=2), **a=56**, **b=78**. Формула виглядає як $56 \cdot 10^2 + 78 = 5600 + 78 = 5678$.

Тепер виразимо добуток **X * Y** через ці частини: $X \cdot Y = (a \cdot 10^{n/2} + b) \cdot (c \cdot 10^{n/2} + d) = (a \cdot c) \cdot 10^n + (a \cdot d + b \cdot c) \cdot 10^{n/2} + (b \cdot d)$

Крок 2: Наївний «Володарюй». Щоб обчислити цей вираз, нам потрібно виконати чотири рекурсивні множення чисел половинної довжини: **a*c**, **a*d**, **b*c** та **b*d**. Складність такого підходу описується рекурентним співвідношенням $T(n) = 4T(n/2) + O(n)$, де $O(n)$ — вартість додавань та зсувів. На жаль, за майстер-теоремою, розв'язком цього співвідношення є $O(n^2)$. Ми не отримали жодного асимптотичного прискорення.

1.3. Прорив швидкодії: від 4 до 3 множень

У 1960 році молодий математик Анатолій Карацуба, працюючи на семінарі, зробив відкриття. Він помітив, що середній член $(a \cdot d + b \cdot c)$ можна обчислити хитріше, уникнувши двох окремих множень.

Ідея Карацуби: Нам у будь-якому разі потрібно обчислити добутки **a*c** та **b*d**. Давайте позначимо їх:

- $z_2 = a \cdot c$
- $z_0 = b \cdot d$

Тепер обчислимо лише один, третій, проміжний добуток:

- $z_1 = (a + b) \cdot (c + d)$

Розкриємо дужки для z_1: $z_1 = a \cdot c + a \cdot d + b \cdot c + b \cdot d$

Подивіться уважно! Цей вираз містить потрібний нам середній член ($a \cdot d + b \cdot c$). Щоб його «дістати», достатньо відняти від z_1 те, що ми вже й так обчислили, — $a \cdot c$ та $b \cdot d$: $a \cdot d + b \cdot c = z_1 - z_2 - z_0$

Це і є прорив! Ми замінили дві дорогі операції множення (`a*d` та `b*c`) на одну (`z1`) та дві дешеві операції віднімання.

1.4. Фінальний алгоритм та його аналіз

Крок 3: Комбінуй. Тепер, маючи три результати (`z2`, `z1`, `z0`), ми можемо зібрати фінальну відповідь за допомогою лише додавань та зсувів (множення на степені 10):

$$X \cdot Y = z_2 \cdot 10^n + (z_1 - z_2 - z_0) \cdot 10^{n/2} + z_0$$

Нове рекурентне співвідношення для складності виглядає так: $T(n) = 3T(n/2) + O(n)$

- $3T(n/2)$: вартість трьох рекурсивних викликів для множення чисел половинного розміру.
- $O(n)$: вартість додавань та віднімань n-значних чисел, які ми виконуємо на кожному кроці.

Застосувавши майстер-теорему, ми отримуємо розв'язок: $T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$ Це значно краще, ніж $O(n^2)$!

Псевдокод алгоритму Карацуби:

```
ФУНКЦІЯ Karatsuba(X, Y):
    // Визначаємо розрядність чисел
    n = максимальна_довжина(X, Y)

    // Базовий випадок рекурсії: якщо числа малі,
    // множимо їх стандартним методом, бо це швидше.
    ЯКЩО n < ПОРОГОВЕ_ЗНАЧЕННЯ (напр. 10) ТО
        ПОВЕРНУТИ X * Y // Стандартне множення
    КІНЕЦЬ ЯКЩО

    // Крок 1: Розділяємо числа на половини
    m = ціла_частина(n / 2)

    // a - старші розряди X, b - молодші розряди X
    a = X / 10^m
```

```
    b = X % 10^m

    // c - старші розряди Y, d - молодші розряди Y
    c = Y / 10^m
    d = Y % 10^m

    // Крок 2: Три рекурсивні виклики
    z2 = Karatsuba(a, c)
    z0 = Karatsuba(b, d)
    z1 = Karatsuba(a + b, c + d)

    // Крок 3: Комбінуємо результати
    middle_term = z1 - z2 - z0
    result = (z2 * 10^(2*m)) + (middle_term * 10^m) + z0

    ПОВЕРНУТИ result
КІНЕЦЬ ФУНКЦІЇ
```

2. Область застосування

Алгоритм Карацуби та його більш просунуті аналоги (наприклад, алгоритм Тоома-Кука) є промисловим стандартом для реалізації арифметики великих чисел.

- Криптографія: Сучасні криптографічні системи (RSA, еліптичні криві) покладаються на операції з дуже великими числами (2048 біт і більше).
- Наукові обчислення: У фізиці, астрономії та інших науках, де потрібна висока точність, що перевищує стандартні типи даних.
- Комп'ютерна алгебра: Системи символьних обчислень (Wolfram Mathematica, Maple) використовують швидкі алгоритми множення для роботи з математичними виразами.

3. Приклади застосування

- Бібліотеки для роботи з великими числами: Будь-яка сучасна програмна бібліотека, як-от **GMP** (GNU Multiple Precision Arithmetic Library) або аналогічні в Python, Java, використовує всередині алгоритм Карацуби для множення.
- Обчислення числа π з високою точністю вимагає множення надзвичайно великих чисел, де прискорення Карацуби стає дуже помітним.

Переваги та недоліки

Переваги:

– Асимптотична швидкість: Для достатньо великих чисел алгоритм Карацуби значно швидший за класичний метод.
– Елегантність: Це чудовий приклад того, як нетривіальний погляд на проблему може призвести до значного прориву.

Недоліки:

– Накладні витрати: Рекурсивні виклики та додаткові операції додавання/віднімання створюють накладні витрати. Через це для малих чисел (наприклад, до кількох десятків знаків) «шкільний» метод буде швидшим.
– Складніша реалізація: Порівняно з простим множенням у стовпчик, алгоритм Карацуби потребує більш складної логіки, особливо при роботі з представленням чисел у вигляді масивів.

Контрольні запитання

21. Поясніть, чому стандартний метод множення «у стовпчик» має складність $O(n^2)$.
22. Скільки рекурсивних множень потрібно було б у наївному підході «Розділяй і володарюй» і чому він не дає прискорення?
23. Запишіть та поясніть формулу, яка є ключовою «хитрістю» в алгоритмі Карацуби для обчислення середнього члена добутку.
24. Чому в псевдокоді алгоритму Карацуби є базовий випадок, який повертається до стандартного множення? Що таке «порогове значення»?
25. Опишіть, як зміниться складність $O(n^{1.585})$, якщо б Карацуба знайшов спосіб обійтися лише двома рекурсивними множеннями.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму множення Карацуби за парадигмою «Розділяй і володарюй».

2. Мета

Практичне засвоєння парадигми «Розділяй і володарюй» на прикладі реалізації алгоритму швидкого множення Карацуби. Порівняння його ефективності зі стандартним методом множення для великих чисел.

3. Завдання

Створити програму, яка реалізує два методи множення для великих цілих чисел: стандартний «шкільний» алгоритм та алгоритм Карацуби. Програма повинна дозволяти вводити два великі числа та обчислювати їх добуток обома методами.

4. Опис

Оскільки стандартні типи даних (int, long long) не можуть зберігати дуже великі числа, їх необхідно представити у вигляді масиву цифр або рядка. Усі арифметичні операції (додавання, віднімання, зсуви) потрібно буде реалізувати для цього представлення.

5. Порядок виконання

1. Оберіть представлення великих чисел: Вирішіть, як ви будете зберігати великі числа (напр., `std::vector<int>` або `std::string`).
2. Реалізуйте допоміжні функції: Напишіть функції для додавання та віднімання великих чисел. Це критично важливий етап.
3. Реалізуйте стандартне множення: Напишіть функцію, що реалізує «шкільний» метод множення у стовпчик. Це буде ваш еталон для перевірки коректності.
4. Реалізуйте алгоритм Карацуби: Керуючись псевдокодом, напишіть рекурсивну функцію `Karatsuba`. Зверніть увагу на коректну реалізацію розбиття числа на частини та фінального об'єднання з урахуванням зсувів (множення на степені 10).
5. Створіть інтерфейс: Напишіть функцію `main`, яка запитує у користувача два числа, обчислює їх добуток обома методами та виводить результати для порівняння.
6. (Опціонально) Виміряйте продуктивність: Для дуже великих чисел (сотні або тисячі знаків) додайте вимірювання часу виконання для обох алгоритмів та порівняйте їх.

6. Результат

Повністю функціональний додаток, що коректно множить великі цілі числа двома різними методами.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко реалізовані обидва алгоритми множення та допоміжні арифметичні функції.
2. Продемонструвати роботу додатку:

- Ввести та перемножити два числа середнього розміру (напр., 10-20 знаків), показавши, що результати обох алгоритмів збігаються.
- Ввести та перемножити два дуже великі числа (напр., 100+ знаків).

3. Бути готовим відповісти на питання:
    - Пояснити, як працює ваша реалізація розбиття числа на частини.
    - У чому полягає рекурсія у вашому коді? Який базовий випадок ви обрали і чому?
    - Яка теоретична складність обох реалізованих вами алгоритмів?

Тести

1. Який крок НЕ є частиною парадигми «Розділяй і володарюй»?

a) Розділяй

b) Володарюй

c) Оптимізуй

d) Комбінуй

3. Скільки рекурсивних викликів множення робить алгоритм Карацуби для перемноження двох чисел?

a) 1

b) 2

c) 3

d) 4

4. Яка асимптотична складність алгоритму Карацуби?

a) $O(N)$

b) $O(N\log N)$

c) $O(N^2)$

d) $O(N^{1.585})$

5. Що є базовим випадком рекурсії в алгоритмі Карацуби?

a) Коли одне з чисел дорівнює нулю.

b) Коли числа стають достатньо малими, щоб їх було ефективніше перемножити стандартним методом.

c) Коли довжина чисел стає непарною.

d) В алгоритмі немає базового випадку.

6. Який вираз є «хитрістю» Карацуби, що дозволяє зменшити кількість множень?

a) $(a + c) \cdot (b + d)$

b) $a \cdot d + b \cdot c$

c) $(a + b) \cdot (c + d) - a \cdot c - b \cdot d$

d) $a \cdot c + b \cdot d$

Відповіді: 1-c, 2-c, 3-d, 4-b, 5-c

Ось 10 теоретичних завдань для самостійної роботи, які поглиблюють розуміння саме методу Карацуби. Вони сфокусовані на аналізі, доведенні та порівнянні, а не на програмуванні.

Завдання для самостійної роботи

1. Аналіз складності. Поясніть своїми словами, чому зменшення кількості рекурсивних множень з 4 до 3 є настільки критичним для покращення асимптотичної складності, незважаючи на те, що це додає більше операцій додавання та віднімання.
2. Алгебраїчне доведення. Починаючи з виразу $z_1 = (a + b) \cdot (c + d)$, алгебраїчно виведіть формулу для обчислення середнього члена ($a \cdot d + b \cdot c$), використовуючи лише z_1, $z_2 = a \cdot c$ та $z_0 = b \cdot d$.
3. Рекурентні співвідношення. Запишіть та порівняйте рекурентні співвідношення для обчислювальної складності: а) Наївного рекурсивного методу множення (з 4 множеннями). б) Методу Карацуби (з 3 множеннями). Поясніть, що означає кожен член у цих співвідношеннях.
4. Застосування Майстер-теореми. Розв’яжіть рекурентне співвідношення для алгоритму Карацуби, $T(n) = 3T(n/2) + O(n)$, за допомогою майстер-теореми. Детально вкажіть, який випадок теореми застосовується і як з нього виводиться кінцева складність.
5. Аналіз базового випадку. Обґрунтуйте, чому в реалізації алгоритму Карацуби є порогове значення, після якого відбувається перехід до стандартного множення. Які фактори (наприклад, особливості архітектури процесора, вартість виклику функції) впливають на вибір оптимального порогового значення?

6. Ручне трасування. Проведіть повне ручне трасування алгоритму Карацуби для множення чисел X = 25 та Y = 41. Детально розпишіть: а) Чому дорівнюють `a, b, c, d`. б) Чому дорівнюють `z0, z2`. в) Як обчислюється `z1`. г) Як з них збирається кінцевий результат.
7. Порівняння з методом «Розділяй і володарюй» Штрассена. Напишіть коротке есе, що порівнює ідеї, які лежать в основі алгоритмів Карацуби та Штрассена. Що в них спільного? Які ключові відмінності в їх застосуванні та отриманому прискоренні?
8. Робота в інших системах числення. Опишіть, як зміниться крок «Комбінуй» в алгоритмі Карацуби, якщо великі числа представлені не в десятковій, а у двійковій системі числення. Яка операція замінить множення на 10^n?
9. Узагальнення ідеї (Алгоритм Тоома-Кука). Дослідіть та опишіть, як алгоритм Тоома-Кука (Toom-3) узагальнює ідею Карацуби. Поясніть, на скільки частин він розбиває числа і як це дозволяє ще більше зменшити кількість необхідних множень (з 9 до 5).
10. Межі застосування. Поміркуйте та опишіть сценарій, де, незважаючи на кращу асимптотичну складність, використання алгоритму Карацуби може бути недоцільним або навіть шкідливим для продуктивності системи.

Алгоритм множення Тоома-Кука

1. Теоретичні відомості

1.1. Від Карацуби до загальної ідеї: множення поліномів

Ми бачили, як алгоритм Карацуби прискорив множення, розбивши числа на дві частини і звівши 4 множення до 3. Виникає закономірне питання: а що, якщо розбити числа на три, чотири або більше частин? Чи можна отримати ще більше прискорення? Відповідь — так, і ця ідея лежить в основі сімейства алгоритмів Тоома-Кука (також відомих як алгоритми Тоома).

Фундаментальна ідея полягає в тому, щоб представити великі числа як поліноми (многочлени). Уявіть n-значне число X. Ми можемо розбити його на k блоків по m цифр у кожному ($n \approx k * m$). Тоді X можна представити у вигляді полінома степеня $k - 1$:

$$X \to P_X(z) = x_{k-1}z^{k-1} + \ldots + x_1 z^1 + x_0 z^0$$

Пояснення позначень:

– x_i: Це i-й блок цифр числа X, який ми розглядаємо як коефіцієнт полінома.

– z: Це основа нашої «системи числення блоків». Якщо ми розбиваємо на блоки по m десяткових цифр, то $z = 10^m$.

– $P_X(z)$: Це поліноміальне представлення числа X. Якщо підставити в нього значення z, ми отримаємо початкове число X.

Приклад: Нехай $X = 12345678$. Розділимо його на 3 блоки ($k = 3$) по 3 цифри ($m = 3$ не виходить, візьмемо $m = 2, n = 8, k = 4$ для простоти). $x_3 = 12, x_2 = 34, x_1 = 56, x_0 = 78$. Тоді $P_X(z) = 12z^3 + 34z^2 + 56z + 78$, де $z = 10^2 = 100$.

Тепер задача множення двох чисел X та Y зводиться до множення їхніх поліноміальних представлень $P_X(z)$ та $P_Y(z)$. Результатом буде новий поліном $P_C(z) = P_X(z) \cdot P_Y(z)$, обчисливши значення якого при нашому z ми отримаємо добуток $X \cdot Y$.

1.2. Інтерполяція: ключ до швидкого множення поліномів

Пряме множення поліномів «коефіцієнт на коефіцієнт» є повільним. Але існує елегантний обхідний шлях, що складається з трьох кроків:

Крок 1: Обчислення значень. Замість того, щоб множити поліноми аналітично, ми обчислимо їх значення у кількох простих точках. Важливий факт з алгебри: поліном степеня d однозначно визначається $d + 1$ точками. Якщо ми множимо два поліноми степеня $k - 1$, результуючий поліном матиме степінь $2(k - 1) = 2k - 2$. Отже, для його визначення нам потрібна $2k - 2 + 1 = 2k - 1$ точка. Ми оберемо $2k - 1$ простих цілочисельних точок (наприклад, 0, 1, -1, 2, ...) і обчислимо значення $P_X(z)$ та $P_Y(z)$ в цих точках.

Крок 2: Поточкові множення. Тепер у нас є два набори чисел. Ми попарно перемножуємо їх. Оскільки ми обрали прості точки, ці числа будуть значно меншими за початкові, і для їх множення ми можемо рекурсивно викликати наш же алгоритм. В результаті ми отримуємо $2k - 1$ значень результуючого полінома $P_C(z)$.

Крок 3: Інтерполяція. Маючи $2k - 1$ значень полінома $P_C(z)$, ми можемо відновити його коефіцієнти. Цей процес називається інтерполяцією. Він зводиться до розв'язання системи лінійних рівнянь.

Ми замінили одне складне множення поліномів на $2k - 1$ простіших множень та додаткові операції додавання, віднімання і ділення на константи на етапах обчислення значень та інтерполяції.

1.3. Алгоритм Toom-3: реалізація

Найпопулярнішим варіантом є Toom-3, де ми розбиваємо кожне число на 3 частини ($k = 3$).

Крок 1: Розділяй Розбиваємо числа X та Y на три блоки: x_2, x_1, x_0 та y_2, y_1, y_0. Формуємо поліноми другого степеня:

- $P_X(z) = x_2 z^2 + x_1 z + x_0$
- $P_Y(z) = y_2 z^2 + y_1 z + y_0$ Результуючий поліном $P_C(z)$ матиме степінь 4, тому нам знадобиться $2 \cdot 3 - 1 = 5$ точок.

Крок 2: Обчислення значень (Evaluation) Обчислимо значення P_X та P_Y у п'яти простих точках: 0, 1, -1, 2, та «нескінченність» (умовно).

- $p(0) = x_0$
- $p(1) = x_2 + x_1 + x_0$
- $p(-1) = x_2 - x_1 + x_0$
- $p(2) = 4x_2 + 2x_1 + x_0$
- $p(\infty) = x_2$ (значення на нескінченності дорівнює старшому коефіцієнту) Аналогічно обчислюємо $q(0), q(1), q(-1), q(2), q(\infty)$ для полінома P_Y.

Крок 3: Поточкові множення Виконуємо 5 рекурсивних множень, щоб отримати 5 значень результуючого полінома $P_C(z)$, який ми позначимо як $r(z)$:

- $r(0) = p(0) \cdot q(0)$
- $r(1) = p(1) \cdot q(1)$
- $r(-1) = p(-1) \cdot q(-1)$
- $r(2) = p(2) \cdot q(2)$
- $r(\infty) = p(\infty) \cdot q(\infty)$

Крок 4: Інтерполяція Ми маємо 5 значень $r(z)$ для полінома $P_C(z) = c_4 z^4 + c_3 z^3 + c_2 z^2 + c_1 z + c_0$. Наше завдання — знайти коефіцієнти c_0, c_1, c_2, c_3, c_4. Це робиться шляхом розв'язання системи з 5 лінійних рівнянь. На щастя, для фіксованих точок (0, 1, -1, 2, ∞) розв'язок є константним і не потребує складних обчислень щоразу. Коефіцієнти знаходяться так:

- $c_0 = r(0)$
- $c_4 = r(\infty)$
- $c_3 = (r(2) - r(-1))/6 - (r(1) - r(-1))/2 - 2c_4$

- $c_1 = (r(1) - r(-1))/2 - c_3$
- $c_2 = r(1) - c_4 - c_3 - c_1 - c_0$ *...(формули можуть дещо відрізнятися в різних реалізаціях, але суть — це лінійні комбінації значень $r(z)$).

Крок 5: Композиція Збираємо фінальну відповідь, підставляючи коефіцієнти в поліном та обчислюючи його значення при нашій базі z: Результат $= c_4 \cdot z^4 + c_3 \cdot z^3 + c_2 \cdot z^2 + c_1 \cdot z + c_0$ Множення на степені z є швидкими операціями зсуву.

1.4. Псевдокод та аналіз складності

```
ФУНКЦІЯ ToomCook3(X, Y):
    n = максимальна_довжина(X, Y)
    // Базовий випадок: для невеликих чисел використовуємо Карацубу а
бо стандартний метод.
    ЯКЩО n < ПОРОГОВЕ_ЗНАЧЕННЯ_TOOM ТО
        ПОВЕРНУТИ Karatsuba(X, Y)
    КІНЕЦЬ ЯКЩО

    // Крок 1: Розділяємо числа на 3 частини.
    m = ціла_частина(n / 3)
    x2, x1, x0 = Розділити(X, m)
    y2, y1, y0 = Розділити(Y, m)

    // Крок 2: Обчислення значень в 5 точках для P_X та P_Y.
    p0 = x0; q0 = y0
    p1 = x2 + x1 + x0; q1 = y2 + y1 + y0
    p_neg1 = x2 - x1 + x0; q_neg1 = y2 - y1 + y0
    p2 = 4*x2 + 2*x1 + x0; q2 = 4*y2 + 2*y1 + y0
    p_inf = x2; q_inf = y2

    // Крок 3: 5 рекурсивних множень.
    r0 = ToomCook3(p0, q0)
    r1 = ToomCook3(p1, q1)
    r_neg1 = ToomCook3(p_neg1, q_neg1)
    r2 = ToomCook3(p2, q2)
    r_inf = ToomCook3(p_inf, q_inf)

    // Крок 4: Інтерполяція для знаходження коефіцієнтів c_i.
    c0 = r0
    c4 = r_inf
    // ... обчислення c1, c2, c3 за формулами
    // (використовуючи r0, r1, r_neg1, r2, c4) ...

    // Крок 5: Композиція результату.
```

```
    Результат = (c4 * z^(4*m)) + (c3 * z^(3*m)) + (c2 * z^(2*m)) + (c
1 * z^m) + c0
    ПОВЕРНУТИ Результат
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності: Ми звели задачу множення чисел довжиною n до 5 множень чисел довжиною $n/3$.

Рекурентне співвідношення: $T(n) = 5T(n/3) + O(n)$ Розв'язком є $O(n^{\log_3 5}) \approx O(n^{1.465})$, що є кращим за $O(n^{1.585})$ у Карацуби.

2. Область застосування

Алгоритм Тоома-Кука використовується в тих же сферах, що і Карацуби, але для чисел ще більшого розміру.

- Криптографія та арифметика великих чисел: Це стандарт де-факто у високопродуктивних бібліотеках.
- Символьні обчислення: Для точних розрахунків у математичних пакетах.
- Теоретична інформатика: Як приклад алгоритму, що дозволяє наблизитись до лінійно-логарифмічної складності.

3. Приклади застосування

- Бібліотека GMP (GNU Multiple Precision Arithmetic Library): Одна з найвідоміших бібліотек для роботи з великими числами. Вона автоматично перемикається між алгоритмами: для малих чисел використовує стандартне множення, для середніх — Карацубу, а для великих (тисячі знаків) — Toom-3. Для астрономічних чисел вона використовує ще швидші методи на основі FFT.
- Python: Вбудована в Python підтримка великих цілих чисел з версії 3 використовує алгоритм Карацуби та, для дуже великих чисел, Тоома-Кука.

Переваги та недоліки

Переваги:

- Краща асимптотична складність: Швидший за алгоритм Карацуби для достатньо великих чисел.
- Узагальнюваність: Метод можна узагальнити до Toom-k, розбиваючи числа на k частин, що дозволяє досягти складності, довільно близької до $O(n^{1+\epsilon})$.

Недоліки:

– Дуже високі накладні витрати: Етапи обчислення значень та інтерполяції є значно складнішими, ніж у Карацуби. Через це Toom-3 стає швидшим за Карацубу лише для чисел, що мають сотні, а то й тисячі знаків.
– Надзвичайна складність реалізації: Коректно реалізувати етап інтерполяції, особливо з урахуванням роботи з від'ємними числами та оптимізацій, є дуже складним завданням.

Контрольні запитання

1. Поясніть своїми словами, чому множення чисел можна звести до множення поліномів.
2. Скільки точок потрібно для однозначного визначення полінома, що є результатом множення двох поліномів степеня *k-1*? Чому?
3. Назвіть три основні етапи алгоритму Тоома-Кука. Який з них є найскладнішим для реалізації?
4. Чому Toom-3 асимптотично швидший за Карацубу? Порівняйте їх рекурентні співвідношення.
5. Чому, незважаючи на кращу асимптотичну складність, алгоритм Карацуби все ще широко використовується і не був повністю замінений на Toom-3?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму множення Тоома-Кука (Toom-3).

2. Мета

Дослідити узагальнення методу Карацуби на прикладі алгоритму Toom-3. Набути навичок реалізації складних рекурсивних алгоритмів та зрозуміти практичні аспекти їх продуктивності.

3. Завдання

(Завдання підвищеної складності). Створити програму, яка реалізує алгоритм множення Toom-3 для великих цілих чисел. Порівняти його роботу з реалізацією алгоритму Карацуби з попередньої роботи.

4. Опис

Це складне завдання, що вимагає уважності. Основна складність полягає не в рекурсивних викликах, а в коректній реалізації етапів обчислення значень та, особливо, інтерполяції, яка включає ділення на константи.

5. Порядок виконання

1. Використайте попередні напрацювання: Вам знадобляться функції додавання та віднімання великих чисел з лабораторної роботи по Карацубі.
2. Реалізуйте розбиття на 3 частини: Напишіть функцію, що ділить велике число на 3 блоки.
3. Реалізуйте етап обчислення значень (Evaluation): Напишіть код, що обчислює $p(0), p(1), p(-1), p(2), p(\infty)$ для полінома P_X.
4. Зробіть 5 рекурсивних викликів: Викличте вашу функцію множення (наприклад, Карацуби як базовий випадок) для 5 пар значень.
5. Реалізуйте етап інтерполяції: Це найскладніший крок. Уважно реалізуйте формули для обчислення коефіцієнтів $c_0, \ldots, c_4$. Будьте обережні з діленням на ціле число.
6. Реалізуйте етап композиції: Зберіть фінальний результат з коефіцієнтів, використовуючи зсуви (множення на степені бази).
7. Протестуйте: Перевірте коректність роботи вашого алгоритму, порівнюючи результати з алгоритмом Карацуби на числах різної довжини.

6. Результат

Додаток, що коректно множить великі цілі числа за алгоритмом Toom-3.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з детальною реалізацією всіх етапів алгоритму.
2. Продемонструвати роботу додатку, порівнюючи результат роботи Toom-3 та Карацуби на однакових великих числах.
3. Бути готовим відповісти на питання:
    - Пояснити, як працює етап інтерполяції у вашому коді.
    - Які основні складнощі виникли при реалізації?
    - В якому випадку (для чисел якої приблизно довжини) ваш Toom-3 починає працювати швидше за Карацубу?

Тести

4. Скільки рекурсивних викликів множення робить алгоритм Toom-3?

е) 3

f) 9

g) 7

h) 5

4. Яка асимптотична складність алгоритму Toom-3?

i) $O(N^{1.585})$

j) $O(N^{1.465})$

k) $O(N\log N)$

l) $O(N^2)$

5. Основна ідея методу Тоома-Кука полягає у зведенні множення великих чисел до:

m) Множення матриць.

n) Сортування масивів.

o) Множення поліномів через обчислення значень та інтерполяцію.

p) Побудови геш-таблиць.

6. Для однозначного відновлення полінома степеня 4 (результат множення в Toom-3) потрібно знати його значення щонайменше у:

q) 4 точках

r) 3 точках

s) 5 точках

t) 9 точках

7. В ієрархії швидкості для дуже великих чисел, яке твердження є правильним?

u) Карацуба швидший за Toom-3.

v) Стандартний метод швидший за Карацубу.

w) Toom-3 швидший за Карацубу.

x) Всі методи мають однакову швидкість.

Відповіді: 1-d, 2-b, 3-c, 4-c, 5-c

Завдання для самостійної роботи

1. Ручне трасування. Проведіть ручне трасування алгоритму Toom-3 для множення чисел X = 123 та Y = 456. Розпишіть етапи розбиття,

обчислення значень в точках 0, 1, -1, поточкових множень та інтерполяції (знайдіть c_0, c_4).

2. Аналіз Toom-2. Дослідіть та опишіть алгоритм Toom-2. Доведіть, що він є математично ідентичним до алгоритму Карацуби.
3. Вибір точок. Поясніть, чому для обчислення значень поліномів в Toom-3 обирають саме точки 0, 1, -1, 2, ∞. Які переваги дає такий вибір?
4. Складність інтерполяції. Оцініть складність етапу інтерполяції. Чому, незважаючи на наявність додавань, віднімань та ділень, його складність оцінюється як $O(n)$?
5. Загальний Toom-k. Напишіть рекурентне співвідношення для загального алгоритму Toom-k, який розбиває числа на k частин. Якою буде його асимптотична складність?

Алгоритм множення Шенхаге-Штрассена

Ми підходимо до вершини в темі швидкого множення чисел. Цей розділ присвячений одному з найбільш елегантних та теоретично важливих алгоритмів, який довгий час був асимптотично найшвидшим. Він складніший, але ідея, що лежить в його основі, є наріжним каменем сучасної обчислювальної математики.

1. Теоретичні відомості

1.1. Новий погляд на множення: згортка поліномів

Ми вже бачили, що великі числа можна представляти у вигляді поліномів. Наприклад, число $X = 582$ можна уявити як поліном $P_X(z) = 5z^2 + 8z + 2$, де $z = 10$. Множення двох чисел X та Y еквівалентне множенню їхніх поліномів $P_X(z)$ та $P_Y(z)$ та обчисленню результуючого полінома $P_C(z)$ в точці z.

Давайте уважно подивимось на коефіцієнти результуючого полінома $C(x) = A(x) \cdot B(x)$. Якщо $A(x) = a_2x^2 + a_1x + a_0$ та $B(x) = b_2x^2 + b_1x + b_0$, то коефіцієнт при x^2, наприклад, буде $c_2 = a_0b_2 + a_1b_1 + a_2b_0$. Загальна формула для k-го коефіцієнта виглядає так:

$$c_k = \sum_{i=0}^{k} a_i \, b_{k-i}$$

Ця операція, що визначає коефіцієнти добутку двох поліномів, в математиці має спеціальну назву — дискретна згортка (convolution). Наївний розрахунок згортки для поліномів степеня n вимагає близько n^2

операцій, що повертає нас до «шкільної» складності $O(n^2)$. Щоб рухатись далі, потрібен принципово новий інструмент.

1.2. Теорема про згортку та Швидке Перетворення Фур'є (FFT)

Цей інструмент — Швидке Перетворення Фур'є (Fast Fourier Transform, FFT). Воно базується на фундаментальному математичному принципі — теоремі про згортку.

Теорема про згортку (спрощено): Складна операція згортки двох послідовностей в «часовій» області (наші коефіцієнти) стає простим поточковим множенням їхніх образів у «частотній» області.

Це означає, що ми можемо замінити одне складне множення поліномів на такий трикроковий процес:

1. Пряме FFT: За допомогою FFT ми перетворюємо вектори коефіцієнтів поліномів $A(x)$ та $B(x)$ у їхні представлення в частотній області. По суті, це еквівалентно обчисленню значень поліномів у наборі специфічних комплексних точок (коренях з одиниці).
2. Поточкове множення: Ми попарно перемножуємо отримані значення. Це проста операція складності $O(n)$.
3. Зворотне FFT (IFFT): За допомогою зворотного перетворення Фур'є ми перетворюємо результат назад, з частотної області в часову, отримуючи коефіцієнти результуючого полінома $C(x)$.

Сам по собі алгоритм FFT є геніальним застосуванням парадигми «Розділяй і володарюй» і має складність $O(n\log n)$. Таким чином, загальна складність множення стає $O(n\log n)$.

1.3. Проблема точності та модульна арифметика

Здається, рішення знайдено. Але є величезна проблема: класичне FFT працює з комплексними числами та арифметикою з плаваючою комою. Це неминуче призводить до помилок округлення. Для інженерних задач це може бути прийнятним, але для множення цілих чисел нам потрібен абсолютно точний результат.

Рішення цієї проблеми лежить у модульній арифметиці. Ідея полягає в тому, щоб виконувати всі обчислення не над полем комплексних чисел, а над скінченним полем або кільцем цілих чисел за модулем M.

Перетворення Фур'є, адаптоване для модульної арифметики, називається числовим теоретичним перетворенням (Number-Theoretic Transform, NTT).

Воно використовує аналоги коренів з одиниці в скінченних полях і гарантує, що всі обчислення є точними, без будь-яких заокруглень.

1.4. Алгоритм Шенхаге-Штрассена

Алгоритм, запропонований Арнольдом Шенхаге та Фолькером Штрассеном у 1971 році, елегантно поєднує ці ідеї. Це рекурсивний алгоритм, який для множення великих чисел використовує швидке множення поліномів на основі NTT.

Ключові особливості:

- Вибір модуля: Щоб NTT працювало ефективно, модуль M має бути обраний дуже ретельно. Шенхаге і Штрассен запропонували використовувати числа виду $M = 2^k + 1$, де k пов'язане з розрядністю чисел, що перемножуються. Такий вибір значно спрощує та прискорює обчислення в модульній арифметиці.
- Рекурсивна структура: Якщо числа занадто великі для прямого застосування NTT (наприклад, не вміщуються в машинні слова), алгоритм рекурсивно викликає сам себе для виконання поточкових множень на другому кроці.

1.5. Псевдокод та аналіз складності

Повна реалізація алгоритму є надзвичайно складною. Наведемо високорівневий псевдокод, що відображає основну ідею.

'''pseudocode ФУНКЦІЯ SchoenhageStrassenMultiply(X, Y): n = розрядність чисел X, Y

```
// Базовий випадок: для недостатньо великих чисел
// викликаємо Toom-Cook або Карацубу.
ЯКЩО n < ПОРОГОВЕ_ЗНАЧЕННЯ_SS ТО
    ПОВЕРНУТИ ToomCook3(X, Y)
КІНЕЦЬ ЯКЩО

// Крок 1: Розбиваємо числа X та Y на блоки (коефіцієнти поліномів).
A = Розбити_на_блоки(X)
B = Розбити_на_блоки(Y)

// Крок 2: Виконуємо пряме NTT над векторами коефіцієнтів.
// Це переводить нас у "частотну" область.
A_freq = NTT(A)
B_freq = NTT(B)

// Крок 3: Виконуємо поточкове множення в частотній області.
```

```
// Множення виконуються рекурсивно!
ДЛЯ і ВІД 0 ДО розмір(A_freq)-1:
    C_freq[i] = SchoenhageStrassenMultiply(A_freq[i], B_freq[i])
КІНЕЦЬ ДЛЯ

// Крок 4: Виконуємо зворотне NTT для повернення в "часову" область.
C_coeffs = InverseNTT(C_freq)

// Крок 5: Композиція результату - "зшиваємо" коефіцієнти C_coeffs
// з урахуванням переносів, щоб отримати фінальне число.
Результат = Композиція(C_coeffs)

ПОВЕРНУТИ Результат
```
КІНЕЦЬ ФУНКЦІЇ ’’’

Аналіз складності: Складність цього алгоритму є $O(N \cdot \log N \cdot \log(\log N))$, де N — кількість біт у числах.

- N: Основна робота.
- $\log N$: Походить від рекурсивної природи FFT/NTT.
- $\log(\log N)$: Походить від рекурсивних викликів множення всередині основної схеми. Ця складність є квазі-лінійною, тобто майже лінійною, і набагато кращою за степеневі складності алгоритмів Карацуби ($O(N^{1.585})$) та Тоома-Кука ($O(N^{1.465})$).

2. Область застосування

Алгоритм Шенхаге-Штрассена — це «важка артилерія» у світі арифметики. Він застосовується там, де потрібна максимальна продуктивність при роботі з астрономічно великими числами.

- Теоретична інформатика та криптоаналіз.
- Обчислення математичних констант (таких як π або e) з рекордною точністю (мільйони та мільярди знаків).
- Високопродуктивні наукові бібліотеки для спеціалізованих задач.

3. Приклади застосування

- Бібліотека GMP (GNU Multiple Precision Arithmetic Library): Це золотий стандарт для арифметики великих чисел. GMP автоматично обирає алгоритм множення залежно від розміру операндів. Для чисел, що мають десятки або сотні тисяч знаків, вона перемикається на алгоритм Шенхаге-Штрассена.

– Тестування простих чисел: Деякі сучасні алгоритми для пошуку дуже великих простих чисел (наприклад, простих чисел Мерсенна) покладаються на швидке множення.

Переваги та недоліки

Переваги:

– Найкраща асимптотична складність серед «класичних» алгоритмів, що використовуються на практиці. Він був найшвидшим відомим алгоритмом протягом майже 40 років.

Недоліки:

– Надзвичайно високі накладні витрати: Через складність перетворень Фур'є та модульної арифметики, цей алгоритм стає швидшим за Тоом-3 лише для чисел, що мають десятки тисяч (а на практиці, скоріше, сотні тисяч) знаків.

– Неймовірна складність реалізації: Написання ефективної та коректної реалізації з нуля є завданням для цілої команди висококваліфікованих програмістів та математиків.

Контрольні запитання

4. Поясніть своїми словами, що таке дискретна згортка і як вона пов'язана з множенням чисел у поліноміальному представленні.
5. У чому полягає основна ідея теореми про згортку?
6. Чому для точного множення цілих чисел не можна використовувати класичне Швидке Перетворення Фур'є (FFT) і чим його замінюють?
7. Опишіть три основні етапи множення за допомогою FFT/NTT.
8. Що означає квазі-лінійна складність $O(N \cdot \log N \cdot \log(\log N))$ і як вона співвідноситься зі складністю алгоритму Тоома-Кука?

Методичні вказівки до лабораторної роботи

1. Назва

Дослідження та реалізація ключового компонента алгоритму Шенхаге-Штрассена: Швидкого Перетворення Фур'є (FFT).

2. Мета

Зрозуміти та реалізувати рекурсивний алгоритм Швидкого Перетворення Фур'є (варіант Кулі-Тьюкі), який є обчислювальним ядром методу Шенхаге-Штрассена.

3. Завдання

(Завдання підвищеної складності). Створити програму, яка реалізує алгоритм FFT для перетворення вектора коефіцієнтів полінома у його представлення в частотній області. *Примітка: Повна реалізація алгоритму Шенхаге-Штрассена є занадто складною для лабораторної роботи, тому ми фокусуємося на його центральній частині.*

4. Опис

Ви будете працювати з масивами комплексних чисел. Алгоритм Кулі-Тьюкі працює за принципом «Розділяй і володарюй», розбиваючи задачу на дві менші підзадачі для парних та непарних коефіцієнтів.

5. Порядок виконання

9. Налаштуйте середовище: Переконайтесь, що ваша мова програмування підтримує комплексні числа (наприклад, $std::complex$ в C++).
10. Реалізуйте рекурсивну функцію FFT:
    - Базовий випадок: якщо вхідний масив має 1 елемент, повернути його ж.
    - Розділіть вхідний масив коефіцієнтів на два: один для парних індексів, інший для непарних.
    - Зробіть два рекурсивні виклики FFT для цих двох підмасивів.
    - Комбінуйте результати, використовуючи формулу «метелика» FFT, що включає комплексні корені з одиниці (експоненти).
11. Створіть тест: Напишіть функцію *main*, яка створює простий поліном (наприклад, $A(x) = x + 1$), викликає для нього FFT і виводить результат. Перевірте його коректність за допомогою онлайн-калькуляторів.

6. Результат

Додаток, що коректно обчислює Швидке Перетворення Фур'є для заданого вектора коефіцієнтів.

7. Після виконання лабораторної роботи студент повинен надати:

12. Вихідний код програми з реалізацією алгоритму FFT.
13. Продемонструвати роботу додатку на кількох тестових поліномах.

14. Бути готовим відповісти на питання:

- Пояснити рекурсивну логіку вашого коду.
- Що таке «корінь з одиниці» і яка його роль у вашому алгоритмі?
- Яка обчислювальна складність вашої реалізації FFT?

Тести

15. Множення чисел в алгоритмі Шенхаге-Штрассена зводиться до:

y) Множення матриць.

z) Дискретної згортки коефіцієнтів поліномів.

aa) Побудови бінарного дерева.

bb) Жадібного вибору.

5. Який математичний інструмент дозволяє обчислити згортку за час $O(n\log n)$?

cc) Метод Монте-Карло.

dd) Алгоритм Дейкстри.

ee) Швидке Перетворення Фур'є (FFT).

ff) Метод Ньютона.

6. Яка головна проблема використання класичного FFT для точного множення цілих чисел?

gg) Висока обчислювальна складність.

hh) Помилки округлення через арифметику з плаваючою комою.

ii) Неможливість роботи з від'ємними числами.

jj) Великі вимоги до пам'яті.

7. Числовим теоретичним перетворенням (NTT) називають FFT, що виконується…

kk) Лише для простих чисел.

ll) В полі комплексних чисел.

mm) В кільці цілих чисел за модулем.

nn) Лише для матриць.

8. Яка асимптотична складність алгоритму Шенхаге-Штрассена?

oo) $O(N^2)$

pp) $O(N^{1.585})$

qq) $O(N\log N)$

rr) $O(N \cdot \log N \cdot \log(\log N))$

Відповіді: 1-b, 2-c, 3-b, 4-c, 5-d

Завдання для самостійної роботи

1. Ручне обчислення згортки: Обчисліть вручну дискретну згортку двох послідовностей: A = {1, 2, 3} та B = {4, 5}.
2. Порівняння FFT та NTT: Напишіть коротке есе, що пояснює ключові відмінності між Швидким Перетворенням Фур'є та Числовим Теоретичним Перетворенням.
3. Вибір модуля для NTT: Дослідіть, чому числа виду $2^k + 1$ (числа Ферма) є зручними для реалізації NTT.
4. Алгоритм Кулі-Тьюкі: Дослідіть та намалюйте схему («метелик») для алгоритму FFT Кулі-Тьюкі при N=8.
5. Корінь з одиниці: Поясніть, що таке первісний корінь n-го степеня з одиниці в модульній арифметиці.
6. Ієрархія алгоритмів: Опишіть, за яким принципом бібліотека типу GMP обирає, який алгоритм множення (стандартний, Карацуби, Тоома-Кука, Шенхаге-Штрассена) використовувати для чисел різної довжини.
7. Роль зворотного FFT: Поясніть, яка роль зворотного перетворення Фур'є (IFFT) у загальному процесі множення. Який етап (обчислення значень чи інтерполяція) воно реалізує?
8. Алгоритм Фюрера: Дослідіть та коротко опишіть, як алгоритм Фюрера (Fürer's algorithm) покращує складність Шенхаге-Штрассена (теоретично).
9. Приклад помилки округлення: Наведіть гіпотетичний приклад, де використання FFT з плаваючою комою може призвести до неправильного результату при множенні двох невеликих цілих чисел.
10. Паддинг: Поясніть, чому перед застосуванням FFT для обчислення згортки вхідні послідовності коефіцієнтів потрібно «доповнювати нулями» (padding) до певної довжини.
11. Теоретична межа: Яка теоретична нижня межа складності для множення двох n-бітних чисел? Чи досягнута вона на практиці?

12. Китайська теорема про залишки: Дослідіть, як Китайська теорема про залишки може використовуватися для реалізації множення великих чисел, коли неможливо знайти один великий модуль для NTT.
13. Практичні реалізації: Знайдіть та проаналізуйте вихідний код реалізації NTT у будь-якій відкритій криптографічній або математичній бібліотеці.
14. Альтернативи: Чи існують інші методи швидкого множення, що не базуються на FFT? Опишіть один, якщо знайдете.
15. Есе-порівняння: Напишіть есе на тему «Практичні компроміси між асимптотичною складністю та накладними витратами», використовуючи всі чотири розглянуті алгоритми множення як приклади.

Алгоритм множення матриць Штрассена

Цей розділ присвячений ще одному революційному застосуванню парадигми «Розділяй і володарюй», яке змінило погляд на ефективність однієї з фундаментальних операцій в обчислювальній математиці.

1. Теоретичні відомості

1.1. Постановка задачі та класичний підхід

Множення матриць — одна з найпоширеніших операцій у комп'ютерній графіці, наукових симуляціях та машинному навчанні. Перш ніж шукати шляхи для оптимізації, давайте згадаємо, як ця операція працює за класичним, або «шкільним», методом.

Припустимо, ми маємо дві квадратні матриці, A та B, розміром $n \times n$. Їхнім добутком буде нова матриця C, також розміром $n \times n$. Кожен елемент матриці C, що знаходиться в i-му рядку та j-му стовпці, позначається як C_{ij} і обчислюється за наступною формулою:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

Пояснення позначень:

- C_{ij}: Це один-єдиний елемент результуючої матриці C. Індекс i вказує на номер рядка, а j — на номер стовпця.

– $\sum_{k=1}^{n}$: Цей математичний символ означає суму. Він каже нам, що ми будемо послідовно змінювати індекс *k* від 1 до *n* і додавати результати.
– $A_{ik} \cdot B_{kj}$: У цьому циклі ми беремо елементи з i-го рядка матриці A та j-го стовпця матриці B. Індекс k, що змінюється, дозволяє нам пройтись по всіх елементах цього рядка та стовпця. Ми попарно множимо елементи ($A_{i1} \cdot B_{1j}$, потім $A_{i2} \cdot B_{2j}$, і так далі) та сумуємо ці добутки.

Аналіз складності. Щоб обчислити один елемент C_{ij}, нам потрібно зробити *n* операцій множення та *n-1* операцію додавання. Оскільки в результуючій матриці C є n^2 елементів, загальна кількість операцій множення становить $n^2 \cdot n = n^3$. Саме тому обчислювальна складність цього алгоритму оцінюється як $O(n^3)$. Довгий час це вважалося теоретичною межею.

1.2. Застосування «Розділяй і володарюй»: перша спроба

Як ми вже бачили на прикладі алгоритму Карацуби, розбиття великої задачі на менші однотипні підзадачі іноді дозволяє досягти асимптотичного прискорення. Спробуємо застосувати цей підхід до множення матриць.

Крок 1: Розділяй. Розділимо кожну з наших матриць A, B та C на чотири рівні підматриці (блоки) розміром $n/2 \times n/2$. Припустимо, що *n* є степенем двійки для простоти.

Крок 2: Наївний «Володарюй». Тепер ми можемо виразити блоки результуючої матриці C через блоки матриць A та B, ніби це звичайні числа. Стандартні правила множення матриць дають нам такі співвідношення:

– $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
– $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
– $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
– $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Давайте проаналізуємо, що ми отримали. Щоб обчислити всі чотири блоки матриці C, нам потрібно виконати 8 рекурсивних множень матриць розміром $n/2 \times n/2$ та 4 додавання матриць такого ж розміру. Складність цього алгоритму описується рекурентним співвідношенням:

$$T(n) = 8T(n/2) + O(n^2)$$

Тут $8T(n/2)$ — це вартість восьми рекурсивних викликів для підматриць половинного розміру, а $O(n^2)$ — вартість чотирьох додавань матриць (додавання двох матриць $m \times m$ вимагає m^2 операцій). Застосувавши

майстер-теорему, ми побачимо, що розв'язком цього співвідношення є $O(n^3)$. Отже, ми не отримали жодного прискорення.

1.3. Прорив Штрассена: від 8 до 7 множень

Протягом багатьох років вважалося, що $O(n^3)$ — це теоретичний мінімум. Однак у 1969 році німецький математик Фолькер Штрассен опублікував статтю, де показав, що це не так. Його ідея, як і в Карацуби, полягала у зменшенні кількості рекурсивних викликів (множень) за рахунок збільшення кількості «дешевших» операцій (додавань та віднімань).

Штрассен знайшов геніальний спосіб обчислити всі чотири блоки C_{ij}, виконавши лише сім множень замість восьми.

Алгоритм Штрассена крок за кроком:

26. Обчислюємо 7 проміжних матриць $P_1 \dots P_7$: Це серце алгоритму. Ми створюємо специфічні комбінації з блоків матриць A і B.

- $P_1 = A_{11} \cdot (B_{12} - B_{22})$
- $P_2 = (A_{11} + A_{12}) \cdot B_{22}$
- $P_3 = (A_{21} + A_{22}) \cdot B_{11}$
- $P_4 = A_{22} \cdot (B_{21} - B_{11})$
- $P_5 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$
- $P_6 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$
- $P_7 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$

6. Комбінуємо їх, щоб отримати фінальний результат: Хоча ці формули виглядають неінтуїтивно, можна алгебраїчно довести, що їх комбінація за допомогою додавань та віднімань точно відтворює блоки результуючої матриці C:

- $C_{11} = P_5 + P_4 - P_2 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_3 + P_4$
- $C_{22} = P_5 + P_1 - P_3 - P_7$

1.4. Аналіз складності алгоритму Штрассена

Завдяки цьому трюку рекурентне співвідношення для складності змінюється:

$$T(n) = 7T(n/2) + O(n^2)$$

– $7T(n/2)$: вартість семи рекурсивних викликів множення для матриць розміром $n/2 \times n/2$.
– $O(n^2)$: вартість 18 операцій додавання/віднімання матриць розміром $n/2 \times n/2$, які ми виконуємо на кожному кроці для обчислення проміжних та фінальних блоків.

Застосовуючи майстер-теорему, ми отримуємо розв'язок:

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.807})$$

Це асимптотично значно краще, ніж $O(n^3)$!

1.5. Псевдокод та деталі реалізації

```
// Функція множення матриць A та B за алгоритмом Штрассена
ФУНКЦІЯ StrassenMultiply(A, B):
    n = розмірність матриці A

    // Базовий випадок: якщо матриці стали достатньо малими,
    // використовувати звичайне множення вигідніше через накладні вит
рати рекурсії.
    ЯКЩО n <= ПОРОГОВЕ_ЗНАЧЕННЯ (напр. 16) ТО
        ПОВЕРНУТИ StandardMultiply(A, B)
    КІНЕЦЬ ЯКЩО

    // Крок 1: Розділяємо матриці A та B на 4 підматриці (блоки) кожн
а.
    A11, A12, A21, A22 = Розділити(A)
    B11, B12, B21, B22 = Розділити(B)

    // Крок 2: Обчислюємо 7 проміжних матриць P, роблячи 7 рекурсивни
х викликів.
    P1 = StrassenMultiply(A11, B12 - B22)
    P2 = StrassenMultiply(A11 + A12, B22)
    P3 = StrassenMultiply(A21 + A22, B11)
    P4 = StrassenMultiply(A22, B21 - B11)
    P5 = StrassenMultiply(A11 + A22, B11 + B22)
    P6 = StrassenMultiply(A12 - A22, B21 + B22)
    P7 = StrassenMultiply(A11 - A21, B11 + B12)

    // Крок 3: Обчислюємо блоки результуючої матриці C, комбінуючи ма
триці P.
    C11 = P5 + P4 - P2 + P6
    C12 = P1 + P2
    C21 = P3 + P4
    C22 = P5 + P1 - P3 - P7
```

```
    // Крок 4: Об'єднуємо блоки C_ij в одну фінальну матрицю C.
    C = Об'єднати(C11, C12, C21, C22)

    ПОВЕРНУТИ C
КІНЕЦЬ ФУНКЦІЇ
```

2. Область застосування

Алгоритм Штрассена та його більш просунуті аналоги є основою для високопродуктивних обчислень у багатьох галузях. Він застосовується там, де необхідне швидке множення великих матриць:

- Наукові та інженерні симуляції: Моделювання фізичних процесів, гідродинаміки, кліматичних змін.
- Комп'ютерна графіка: Застосування матричних перетворень до великої кількості вершин.
- Аналіз даних та машинне навчання: Деякі алгоритми покладаються на операції з великими матрицями.
- Теоретична інформатика: Як будівельний блок для інших швидких алгоритмів, наприклад, для обчислення оберненої матриці.

3. Приклади застосування

- BLAS (Basic Linear Algebra Subprograms): Високо оптимізовані наукові бібліотеки, такі як OpenBLAS або Intel MKL, використовують алгоритми типу Штрассена для множення великих матриць, автоматично обираючи найкращий метод залежно від розміру матриць та архітектури процесора.
- Обробка зображень: Деякі методи фільтрації та розпізнавання образів можуть бути зведені до операцій множення матриць.
- Пошукові системи: Аналіз зв'язків між веб-сторінками (як в алгоритмі PageRank) вимагає операцій з величезними матрицями суміжності.

Переваги та недоліки

Переваги:

- Асимптотична швидкість: Теоретично значно швидший за стандартний метод для великих n. Це був прорив, що довів, що складність $O(n^3)$ не є оптимальною.

Недоліки:

– Високий коефіцієнт прихованої константи: Через велику кількість додавань/віднімань (18 операцій над матрицями розміру n/2) алгоритм Штрассена стає швидшим за звичайний лише для матриць значного розміру (зазвичай, $n > 32$ або $n > 64$).

– Складність реалізації: Алгоритм значно складніший в програмуванні, ніж ітеративний метод. Особливо складною є ефективна реалізація, що враховує кешування та паралелізм.

– Числова нестійкість: Використання віднімань може призводити до більшої втрати точності при роботі з числами з плаваючою комою порівняно зі стандартним алгоритмом.

– Вимоги до пам'яті: Потребує додаткової пам'яті для зберігання проміжних матриць $P_1 \dots P_7$ та матриць, що є результатами додавань/віднімань.

Контрольні запитання

1. Поясніть своїми словами, чому наївний підхід «Розділяй і володарюй» з 8 рекурсивними викликами не дає прискорення порівняно з $O(n^3)$?
2. У чому полягає головний «трюк» алгоритму Штрассена?
3. Чому, незважаючи на кращу асимптотичну складність, алгоритм Штрассена не використовують для множення матриць малого розміру (наприклад, 4x4)?
4. Скільки операцій додавання та віднімання матриць розміру $n/2$ потрібно зробити на кожному кроці рекурсії в алгоритмі Штрассена?
5. Як розмір матриці впливає на вибір між стандартним алгоритмом та алгоритмом Штрассена?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму множення матриць Штрассена.

2. Мета

Практичне засвоєння парадигми «Розділяй і володарюй» на прикладі реалізації алгоритму Штрассена. Аналіз продуктивності та визначення «точки перетину», де алгоритм Штрассена стає ефективнішим за стандартний.

3. Завдання

Створити програму, яка реалізує два методи множення для квадратних матриць $n \times n$ (де n — степінь двійки): стандартний ітеративний алгоритм та рекурсивний алгоритм Штрассена.

4. Опис

Ваша програма буде працювати з двовимірними масивами. Перш ніж реалізовувати сам алгоритм Штрассена, вам знадобляться «будівельні блоки» — допоміжні функції для роботи з матрицями.

5. Порядок виконання

1. Підготуйте інструменти: Створіть допоміжні функції для додавання та віднімання матриць.
2. Реалізуйте «золотий стандарт»: Напишіть функцію, що реалізує звичайне множення ($O(n^3)$). Вона буде потрібна для порівняння та для базового випадку рекурсії.
3. Реалізуйте алгоритм Штрассена: Керуючись псевдокодом, напишіть рекурсивну функцію `StrassenMultiply`.
4. Визначте базовий випадок: Встановіть порогове значення розміру матриці (n), при якому рекурсія зупиняється і викликається стандартне множення. Почніть, наприклад, з `n=2` і експериментуйте.
5. Створіть тест-бенч: Напишіть функцію `main`, яка створює матриці заданого розміру (наприклад, заповнюючи їх випадковими числами), виконує множення обома методами та перевіряє, чи збігаються результати.
6. Проведіть експеримент: Виміряйте час виконання обох алгоритмів для матриць різного розміру (n = 16, 32, 64, 128, 256…).
7. Побудуйте графік залежності часу від розміру та знайдіть точку, де алгоритм Штрассена стає швидшим.

6. Результат

Додаток, що коректно множить матриці двома методами та виводить порівняння часу їх виконання.

10. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко реалізовані обидва алгоритми множення та допоміжні функції.
2. Продемонструвати роботу додатку:

- Показати коректність роботи на матрицях малого розміру.
- Показати результати вимірювання часу для великих матриць та визначену «точку перетину» ефективності.

3. Бути готовим відповісти на питання:
    - Пояснити, як у вашому коді реалізовано розбиття на підматриці та їх об’єднання.
    - Чому ви обрали саме таке порогове значення для базового випадку рекурсії?
    - Які практичні складнощі виникли при реалізації алгоритму Штрассена?

Тести

1. Скільки операцій множення скалярних чисел потребує стандартний алгоритм для перемноження двох матриць 2x2?

ss) 4

tt) 7

uu) 8

vv) 18

2. Яка асимптотична складність алгоритму Штрассена?

ww) $O(n^2)$

xx) $O(n\log n)$

yy) $O(n^3)$

zz) $O(n^{\log_2 7})$

3. Ключова ідея Штрассена полягає у зменшенні кількості…

aaa) Рекурсивних додавань матриць з 8 до 7.

bbb) Рекурсивних множень матриць з 8 до 7.

ccc) Рекурсивних віднімань матриць з 10 до 8.

ddd) Усіх операцій загалом.

4. Чому алгоритм Штрассена зазвичай не використовують для малих матриць?

eee) Він дає неправильний результат для матриць менше 16x16.

fff) Він вимагає занадто багато пам’яті.

ggg) Він не працює з цілими числами.

hhh) Великі накладні витрати на рекурсію та додаткові операції додавання/віднімання роблять його повільнішим.

5. Що є базовим випадком рекурсії в алгоритмі Штрассена?

iii) Коли матриця складається з одного елемента.

jjj) Коли розмір матриці досягає певного порогового значення, і стає вигідніше використати стандартний алгоритм.

kkk) Коли всі елементи матриці стають нульовими.

lll) Алгоритм не має базового випадку.

Відповіді: 1-c, 2-d, 3-b, 4-d, 5-b

Завдання для самостійної роботи

1. Ручне трасування. Вручну обчисліть добуток двох матриць 2x2 за алгоритмом Штрассена. Запишіть всі 7 проміжних матриць P_i (у цьому випадку вони будуть просто числами) та 4 результуючі елементи C_{ij}.
2. Аналіз операцій. Доведіть, що на кожному кроці рекурсії алгоритм Штрассена виконує рівно 18 операцій додавання/віднімання матриць розміру n/2.
3. Нестепеневі двійки. Детально опишіть алгоритм, як можна модифікувати реалізацію Штрассена для множення матриць розміром $n \times n$, де n не є степенем двійки.
4. Множення прямокутних матриць. Дослідіть, чи можна адаптувати алгоритм Штрассена для множення прямокутних матриць (наприклад, A розміром $m \times k$ та B розміром $k \times n$). Якщо так, то як?
5. Алгоритм Штрассена-Віноградa. Дослідіть та опишіть модифікацію алгоритму, відому як Штрассена-Віноградa, яка вимагає 15 додавань/віднімань замість 18.
6. Числова стійкість. Напишіть коротке есе, що пояснює, чому велика кількість операцій віднімання в алгоритмі Штрассена може призводити до накопичення помилок при роботі з числами з плаваючою комою.
7. Обернення матриці. Дослідіть, як алгоритм Штрассена використовується для прискорення алгоритму обернення матриці. (Підказка: це пов’язано з блочним методом обернення).

8. Порівняння з Coppersmith–Winograd. Знайдіть інформацію про алгоритм Копперсміта-Віноградa. Яка його теоретична складність і чому він не використовується на практиці?
9. Оптимізація пам’яті. Запропонуйте стратегію, як можна зменшити використання додаткової пам’яті в реалізації алгоритму Штрассена.
10. Теоретичне обґрунтування. Спробуйте алгебраїчно довести коректність формули для одного з результуючих блоків, наприклад, для $C_{12} = P_1 + P_2$.
11. Реалізація допоміжних функцій. Напишіть псевдокод для функцій `Розділити(M)` та `Об'єднати(M11, M12, M21, M22)`, які використовуються в основному алгоритмі.
12. Аналіз рекурсії. Побудуйте дерево рекурсивних викликів для алгоритму Штрассена для матриці 4x4. Порахуйте загальну кількість множень 2x2, які будуть виконані.
13. Кеш-ефективність. Порівняйте стандартний ітеративний алгоритм множення та рекурсивний алгоритм Штрассена з точки зору кеш-ефективності. Який з них краще використовує кеш-пам’ять процесора і чому?
14. Альтернативне розбиття. Чи існують алгоритми типу Штрассена, що розбивають матрицю не на 4, а на 9 блоків (3x3)? Дослідіть це питання.
15. Есе-роздум. Напишіть есе на тему: «Межі практичного застосування асимптотично ефективних алгоритмів на прикладі множення матриць».

Жадібні алгоритми

Ми переходимо до нової фундаментальної парадигми. Розділ побудований так, щоб провести студента від інтуїтивного розуміння до глибокого аналізу та практичного застосування жадібних алгоритмів.

1. Теоретичні відомості

1.1. Ідея жадібного підходу: локальна вигода, глобальна мета

Уявіть, що вам потрібно дати решту, скажімо, 48 гривень, використовуючи якомога менше монет. У вас є монети номіналом 10, 5, 2 та 1 грн. Не довго

думаючи, ви робите наступне: берете найбільшу можливу монету (10 грн), потім ще одну (10), і так, доки це можливо. Коли 10 грн вже забагато, ви берете найбільшу з решти (5 грн), і так далі. На кожному кроці ви робите локально оптимальний вибір — берете найціннішу монету, яка не перевищує залишок. В кінці ви отримуєте глобально оптимальний результат — мінімальну кількість монет.

Це і є суть жадібного алгоритму. Це парадигма, в якій для розв'язання задачі ми на кожному етапі робимо вибір, який здається найкращим у даний момент, не заглядаючи у майбутнє і не переглядаючи попередні рішення. Ми сподіваємось, що послідовність таких локально оптимальних кроків приведе нас до глобального оптимуму.

1.2. Коли працює жадібність? Ключові властивості

На жаль, такий прямолінійний підхід працює не завжди. Якби у вас були монети 1, 6 та 10 грн, а решту треба було дати 12 грн, жадібний алгоритм взяв би монету 10, а потім дві по 1 (всього 3 монети). Хоча оптимальний розв'язок — дві монети по 6 грн.

Щоб жадібний алгоритм гарантовано давав правильний, оптимальний розв'язок, задача повинна мати дві важливі властивості:

1. Властивість жадібного вибору. Це означає, що глобально оптимальний розв'язок можна отримати, зробивши локально оптимальний вибір. Іншими словами, вибір, зроблений на поточному кроці, не закриває шлях до кінцевого оптимуму. Нам не потрібно «озиратися назад» і переглядати попередні рішення.
2. Властивість оптимальної підструктури. Ця властивість означає, що оптимальний розв'язок всієї задачі містить у собі оптимальні розв'язки для її підзадач. Наприклад, якщо ми зробили перший жадібний вибір, то задача, що залишилася (дати решту грошей), також має бути розв'язана оптимально, щоб кінцевий результат був оптимальним.

Примітка: Ця властивість є спільною для жадібних алгоритмів та динамічного програмування, але на відміну від ДП, жадібний підхід не розглядає всі можливі розв'язки підзадач, а одразу робить один вибір.

1.3. Детальний розбір: Задача про вибір заявок

Це класична задача, що ідеально ілюструє жадібний підхід. Постановка задачі: Є множина з n заявок (або лекцій, подій), кожна з яких має час

початку s_i та час закінчення f_i. Дві заявки є сумісними, якщо їхні інтервали часу не перетинаються. Потрібно обрати максимальну кількість попарно сумісних заявок.

Пошук жадібної стратегії. Який вибір буде локально оптимальним? Спробуємо декілька ідей:

- *Вибирати найкоротшу заявку?* Не спрацює. Коротка заявка може перетинатися з двома довгими, які могли б бути обрані замість неї.
- *Вибирати заявку, що починається найраніше?* Теж ні. Рання, але дуже довга заявка може завадити проведенню кількох коротших.
- Правильна стратегія: вибирати заявку, що закінчується найраніше. Інтуїція підказує, що обираючи заявку, яка завершується якнайшвидше, ми звільняємо наш ресурс (наприклад, лекційну аудиторію) для наступних подій якомога раніше.

Доведення коректності:

1. Властивість жадібного вибору: Нехай заявка a_1 — це заявка з найранішим часом закінчення. Припустимо, існує оптимальний розв'язок, який не включає a_1. Нехай перша заявка в цьому розв'язку — це a_k. Оскільки a_1 закінчується раніше за будь-яку іншу заявку, то час її закінчення $f_1 \leq f_k$. Ми можемо безболісно замінити a_k на a_1 в оптимальному розв'язку. a_1 не буде конфліктувати з наступними заявками, бо вона закінчується раніше, ніж a_k. Таким чином, ми отримали новий оптимальний розв'язок, який починається з нашого жадібного вибору.
2. Властивість оптимальної підструктури: Зробивши перший вибір (взявши a_1), ми стикаємося з підзадачею: знайти максимальну кількість сумісних заявок серед тих, що починаються після закінчення a_1. Якщо ми розв'яжемо цю підзадачу оптимально, то, поєднавши її розв'язок з a_1, ми отримаємо глобальний оптимум.

1.4. Псевдокод алгоритму

```
// Вхід: масив 'activities' з об'єктів, де кожен має поля 'start' та
'finish'
ФУНКЦІЯ GreedyActivitySelector(activities):
    // Крок 1: Сортуємо заявки за зростанням часу їх закінчення.
    Sort(activities, за полем 'finish')

    // Ініціалізуємо результат першою заявкою.
```

```
    // Вона гарантовано є частиною одного з оптимальних розв'язків.
    n = кількість(activities)
    ЯКЩО n == 0 ТО ПОВЕРНУТИ пустий_список КІНЕЦЬ ЯКЩО

    result = [activities[0]] // Додаємо першу заявку до результату
    last_finish_time = activities[0].finish

    // Крок 2: Проходимо по решті заявок.
    ДЛЯ i ВІД 1 ДО n-1:
        // Якщо поточна заявка починається після закінчення останньої
обраної,
        // вона є сумісною.
        ЯКЩО activities[i].start >= last_finish_time ТО
            // Робимо жадібний вибір - додаємо цю заявку до результат
у.
            Додати(result, activities[i])
            // Оновлюємо час закінчення останньої обраної заявки.
            last_finish_time = activities[i].finish
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    ПОВЕРНУТИ result
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності: Найдорожчою операцією є сортування, яке має складність $O(n\log n)$. Наступний цикл виконується за лінійний час $O(n)$. Отже, загальна складність алгоритму — $O(n\log n)$.

2. Область застосування

Жадібні алгоритми застосовуються для розв’язання оптимізаційних задач, де потрібно знайти найкращий (максимальний або мінімальний) результат.

- Алгоритми на графах: Пошук найкоротших шляхів (Дейкстра), побудова мінімальних кістякових дерев (Прим, Крускал).
- Стиснення даних: Алгоритм Гаффмана для побудови оптимальних префіксних кодів.
- Планування та розподіл ресурсів: Задача про вибір заявок, задача про дрібний рюкзак.

3. Приклади застосування

- Мережеві маршрутизатори: Використовують алгоритми, подібні до Дейкстри, щоб знайти найкоротший шлях для передачі пакетів даних.

– Планувальники завдань в ОС: Можуть використовувати жадібні стратегії для виконання процесів (наприклад, виконувати найкоротше завдання першим).
– Архіватори (ZIP, GZIP): Використовують кодування Гаффмана, яке є жадібним алгоритмом, для ефективного стиснення файлів.

Переваги та недоліки

Переваги:

– Простота реалізації: Жадібні алгоритми зазвичай простіші для розуміння та реалізації, ніж, наприклад, динамічне програмування.
– Висока швидкість: Часто вони є дуже ефективними, оскільки не потребують розгляду всіх можливих варіантів. Складність зазвичай визначається сортуванням та одним проходом по даних.

Недоліки:

– Обмеженість застосування: Працюють лише для задач, що мають властивості жадібного вибору та оптимальної підструктури.
– Складність доведення коректності: Найскладніша частина — довести, що жадібна стратегія справді призводить до глобального оптимуму для даної задачі.
– Негнучкість: Алгоритм не повертається назад, щоб переглянути свій вибір. Якщо на якомусь кроці було прийнято неоптимальне рішення, помилка не буде виправлена.

Контрольні запитання

1. Сформулюйте основний принцип, що лежить в основі жадібних алгоритмів.
2. Назвіть та поясніть дві властивості, якими має володіти задача, щоб її можна було розв’язати жадібним алгоритмом.
3. Наведіть приклад задачі, де жадібний підхід не дає оптимального розв’язку.
4. Яка жадібна стратегія є правильною для задачі про вибір заявок і чому стратегія «обирати найкоротшу заявку» є хибною?
5. Порівняйте жадібні алгоритми та динамічне програмування. Що в них спільного і в чому ключова відмінність?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація жадібного алгоритму для задачі про вибір заявок.

2. Мета

Практичне засвоєння парадигми жадібних алгоритмів, включаючи сортування даних як підготовчий етап та ітеративний вибір на основі локально оптимального критерію.

3. Завдання

Створити програму, яка приймає на вхід набір заявок (інтервалів часу) та знаходить максимальну кількість сумісних заявок, використовуючи жадібний алгоритм.

4. Опис

Кожна заявка може бути представлена структурою або класом з двома полями: `start_time` та `finish_time`. Програма повинна реалізувати логіку, описану в псевдокоді.

5. Порядок виконання

1. Створіть структуру даних: Опишіть структуру `Activity` для зберігання часу початку та кінця заявки.
2. Підготуйте вхідні дані: Створіть масив або список з декількох об’єктів `Activity` для тестування.
3. Реалізуйте сортування: Відсортуйте масив заявок за зростанням часу їх закінчення (`finish_time`). Можна використовувати вбудовану функцію сортування вашої мови програмування.
4. Реалізуйте жадібний вибір: Керуючись псевдокодом, напишіть функцію `GreedyActivitySelector`. Ініціалізуйте розв’язок першою заявкою з відсортованого масиву.
5. Ітеруйте та обирайте: Пройдіться по решті заявок, порівнюючи час початку кожної з часом закінчення останньої обраної заявки. Якщо вони сумісні, додайте заявку до розв’язку.
6. Виведіть результат: Напишіть функцію `main`, яка викликає ваш алгоритм та виводить на екран список обраних заявок та їхню загальну кількість.

6. Результат

Додаток, що коректно знаходить максимальний набір сумісних заявок для заданого вхідного набору.

7. Після виконання лабораторної роботи студент повинен надати:
    1. Вихідний код програми, де чітко реалізована структура заявки, сортування та жадібний вибір.
    2. Продемонструвати роботу додатку на кількох тестових наборах, включаючи той, де очевидний вибір не є оптимальним.
    3. Бути готовим відповісти на питання:
        - Пояснити, чому сортування за часом закінчення є ключовим для роботи алгоритму.
        - Яка обчислювальна складність вашої програми і який етап є «найдорожчим»?
        - Що станеться, якщо відсортувати заявки за часом початку? Чи буде алгоритм працювати коректно?

Тести

1. Який основний принцип жадібного алгоритму?

mmm) Розв’язати всі підзадачі, а потім обрати найкращий шлях.

nnn) На кожному кроці робити локально оптимальний вибір.

ooo) Перебрати всі можливі варіанти розв’язку.

ppp) Повертатися назад, якщо поточний вибір виявився поганим.

3. Яка з цих задач НЕ розв’язується класичним жадібним підходом?

qqq) Пошук найкоротшого шляху в графі з невід’ємними вагами (алгоритм Дейкстри).

rrr) Задача про вибір заявок.

sss) Задача про рюкзак 0-1 (предмети не можна ділити).

ttt) Побудова коду Гаффмана.

4. Яка асимптотична складність жадібного алгоритму для задачі про вибір заявок?

uuu) $O(n)$

vvv) $O(n^2)$

www) $O(n\log n)$

xxx) $O(\log n)$

5. Правильна жадібна стратегія для задачі про вибір заявок — це обирати заявку, яка:

yyy) Починається найраніше.

zzz) Є найкоротшою.

aaaa) Перетинається з найменшою кількістю інших заявок.

bbbb) Закінчується найраніше.

6. Яка властивість НЕ є обов'язковою для коректності жадібного алгоритму?

cccc) Властивість жадібного вибору.

dddd) Властивість оптимальної підструктури.

eeee) Властивість перекриття підзадач.

ffff) Всі перелічені є обов'язковими.

Відповіді: 1-b, 2-c, 3-c, 4-d, 5-c

Завдання для самостійної роботи

1. Задача про дрібний рюкзак. Є рюкзак з обмеженою вантажопідйомністю W та набір предметів, кожен з яких має вагу та вартість. Предмети можна ділити. Потрібно заповнити рюкзак так, щоб сумарна вартість була максимальною. Реалізуйте жадібний алгоритм для цієї задачі.
2. Доведення від супротивного. Наведіть формальний контрприклад, що доводить, чому стратегія «обирати заявку, що починається найраніше» не є оптимальною для задачі про вибір заявок.
3. Задача про заправки. Автомобіль має проїхати з пункту А в пункт Б. Бак вміщує пального на N кілометрів. На шляху є заправки. Розробіть жадібний алгоритм, що мінімізує кількість зупинок на заправках.
4. Кодування Гаффмана. Дослідіть та реалізуйте алгоритм Гаффмана для стиснення тексту. Поясніть, у чому полягає його «жадібність».
5. Алгоритм Крускала. Реалізуйте алгоритм Крускала для знаходження мінімального кістякового дерева у графі. Яка жадібна стратегія лежить в його основі?
6. Алгоритм Прима. Реалізуйте алгоритм Прима для знаходження мінімального кістякового дерева. Порівняйте його жадібну стратегію зі стратегією алгоритму Крускала.

7. Задача про платформи. На вокзал прибувають та відбувають потяги у визначений час. Знайдіть мінімальну кількість платформ, необхідних для обслуговування всіх потягів.
8. Задача про покриття множини (Set Cover). Дослідіть цю класичну NP-складну задачу. Розробійте жадібний *наближений* алгоритм для неї та поясніть, чому він не гарантує оптимального розв'язку.
9. Задача про монети (неканонічна система). Наведіть приклад набору номіналів монет, для якого жадібний алгоритм видачі решти не працює.
10. Порівняння з ДП. Розгляньте задачу про рюкзак 0-1. Поясніть, чому для неї не працює жадібний підхід, але працює динамічне програмування. У чому принципова відмінність у «прийнятті рішень» цих двох парадигм на прикладі цієї задачі?
11. Планування з дедлайнами. Є набір завдань, кожне з яких має тривалість виконання та крайній термін (дедлайн). Розробіть жадібний алгоритм для планування завдань так, щоб мінімізувати максимальне запізнення.
12. З'єднання файлів. Уявіть, що у вас є багато файлів, і вам потрібно об'єднати їх усі в один шляхом послідовного злиття двох файлів. Вартість злиття двох файлів дорівнює сумі їх розмірів. Розробіть жадібний алгоритм, що мінімізує загальну вартість злиття. (Підказка: це схоже на алгоритм Гаффмана).
13. Розподіл навантаження. Є m однакових процесорів та n завдань з відомим часом виконання кожного. Розробійте жадібний алгоритм, що намагається розподілити завдання між процесорами так, щоб час завершення останнього завдання (makespan) був мінімальним. Чи є цей алгоритм оптимальним?
14. Жадібний алгоритм для розфарбовування графа. Дослідіть простий жадібний алгоритм для розфарбовування вершин графа (послідовно обирати для кожної вершини найменший можливий колір). Наведіть приклад графа, для якого цей алгоритм використовує більше кольорів, ніж мінімально необхідно.
15. Есе-аналіз. Напишіть есе на тему «Чому доведення коректності є найважливішим етапом при розробці жадібного алгоритму?».

Динамічне програмування:

Ми переходимо до однієї з найпотужніших та найцікавіших алгоритмічних парадигм. Динамічне програмування часто здається складним, але його ідея в основі своїй дуже елегантна. Цей розділ побудований так, щоб провести вас від інтуїтивного розуміння проблеми до впевненого володіння цим методом.

1. Теоретичні відомості

1.1. Вступ: неефективність «лобової» рекурсії

Уявімо, що нам потрібно обчислити n-те число Фібоначчі. З математики ми знаємо просте рекурентне співвідношення: $F_n = F_{n-1} + F_{n-2}$, з базовими випадками $F_0 = 0, F_1 = 1$. Написати рекурсивну функцію для цього здається тривіальним:

```
ФУНКЦІЯ fib(n):
    ЯКЩО n <= 1 ТО
        ПОВЕРНУТИ n
    КІНЕЦЬ ЯКЩО
    ПОВЕРНУТИ fib(n-1) + fib(n-2)
КІНЕЦЬ ФУНКЦІЇ
```

Це виглядає просто і красиво, але спробуйте обчислити `fib(40)`. Програма «зависне» надовго. Чому? Давайте подивимось на дерево викликів для `fib(5)`:

Ми бачимо, що `fib(3)` обчислюється двічі, `fib(2)` — тричі, і так далі. З ростом `n` кількість повторних обчислень зростає експоненційно, що призводить до катастрофічної складності $O(2^n)$. Динамічне програмування — це техніка, створена саме для того, щоб елегантно вирішувати такі проблеми.

1.2. Два стовпи динамічного програмування

Щоб до задачі можна було застосувати ДП, вона повинна задовольняти двом ключовим властивостям:

1. Перекриття підзадач. Це саме те, що ми бачили у прикладі з числами Фібоначчі. Рекурсивний розв’язок задачі раз по раз повертається до обчислення одних і тих самих менших підзадач. Динамічне програмування використовує це, обчислюючи кожну підзадачу лише один раз і зберігаючи її результат у пам’яті.
2. Оптимальна підструктура. Ця властивість означає, що оптимальний розв’язок всієї задачі може бути побудований з оптимальних

розв’язків її підзадач. Для чисел Фібоначчі це очевидно: розв’язок для F_n будується з розв’язків для F_{n-1} та F_{n-2}. Ця властивість спільна з жадібними алгоритмами, але, як ми побачимо, ДП використовує її інакше.

1.3. Два підходи до розв’язання: мемоізація та табуляція

Існує два основні методи реалізації ДП.

1. Top-Down з мемоізацією (зверху-вниз). Цей підхід є прямою оптимізацією наївної рекурсії. Ми пишемо ту саму рекурсивну функцію, але додаємо «кеш» або «пам’ятку» (memo), де зберігаємо вже обчислені результати.

   Ідея: Перед тим, як обчислювати, перевір, чи немає відповіді у пам’ятці. Якщо є — візьми її. Якщо немає — обчисли, збережи у пам’ятку, а потім поверни результат.

Псевдокод для Фібоначчі з мемоізацією:

```
// Ініціалізуємо масив 'memo' значеннями, що вказують на необчисленість (напр., -1)
ФУНКЦІЯ fib_memo(n, memo):
    // Якщо значення вже було обчислене, просто повертаємо його
    ЯКЩО memo[n] != -1 ТО
        ПОВЕРНУТИ memo[n]
    КІНЕЦЬ ЯКЩО

    // Базовий випадок
    ЯКЩО n <= 1 ТО
        ПОВЕРНУТИ n
    КІНЕЦЬ ЯКЩО

    // Обчислюємо результат, ЯКЩО його ще немає
    result = fib_memo(n-1, memo) + fib_memo(n-2, memo)

    // Зберігаємо результат у пам'ятку перед поверненням
    memo[n] = result
    ПОВЕРНУТИ result
КІНЕЦЬ ФУНКЦІЇ
```

Складність тепер стає лінійною — $O(n)$, оскільки кожне значення `fib(i)` обчислюється лише один раз.

2. Bottom-Up з табуляцією (знизу-вгору). Цей підхід є ітеративним і часто більш ефективним. Замість того, щоб починати з великої задачі `n` і спускатись, ми починаємо з найменших підзадач і поступово

«піднімаємось» вгору. Ідея: Створимо таблицю (масив) для зберігання розв'язків і заповнимо її, починаючи з базових випадків.

Псевдокод для Фібоначчі з табуляцією:

```
ФУНКЦІЯ fib_tab(n):
    // Створюємо таблицю для зберігання результатів
    dp = новий_масив_розміром(n + 1)

    // Заповнюємо базові випадки
    dp[0] = 0
    dp[1] = 1

    // Заповнюємо решту таблиці, використовуючи попередні результати
    ДЛЯ i ВІД 2 ДО n:
        dp[i] = dp[i-1] + dp[i-2]
    КІНЕЦЬ ДЛЯ

    ПОВЕРНУТИ dp[n]
КІНЕЦЬ ФУНКЦІЇ
```

Цей підхід також має складність $O(n)$, але зазвичай є трохи швидшим на практиці, оскільки не має накладних витрат на рекурсивні виклики.

1.4. Детальний розбір: Задача про рюкзак 0/1

Це класична оптимізаційна задача, яка чудово демонструє силу ДП і на якій «ламаються» жадібні алгоритми.

Постановка задачі: Є набір з n предметів, кожен з яких має вагу w_i та вартість v_i. Також є рюкзак, що може витримати максимальну вагу W. Потрібно обрати такий набір предметів, щоб їхня сумарна вартість була максимальною, а сумарна вага не перевищувала W. Важливо: кожен предмет можна або взяти цілком, або не брати взагалі (властивість «0/1»).

Чому жадібний підхід не працює? Жадібна стратегія (наприклад, «брати найдорожчі предмети» або «брати предмети з найкращим співвідношенням вартість/вага») може призвести до неоптимального розв'язку. Ви можете взяти один дуже дорогий, але важкий предмет, і він займе все місце, хоча можна було б набрати кілька менш дорогих, але легких предметів із більшою сумарною вартістю.

Формулювання рекурентного співвідношення для ДП: Щоб розв'язати задачу, нам потрібно побудувати таблицю `dp[i][j]`, де `dp[i][j]` буде зберігати максимальну вартість, яку можна отримати, використовуючи лише перші `i` предметів при максимальній вазі рюкзака `j`.

Розглядаючи *i*-й предмет, у нас є два варіанти:

1. Не брати *i*-й предмет: Тоді максимальна вартість така ж, як і максимальна вартість для перших *i-1* предметів при тій самій вазі рюкзака `j`. Тобто, `dp[i-1][j]`.
2. Брати *i*-й предмет (якщо його вага $w_i \leq j$): Тоді вартість складатиметься з вартості самого предмета v_i плюс максимальна вартість, яку можна було набрати з перших *i-1* предметів у рюкзак із залишеною вагою $j - w_i$. Тобто, $v_i + dp[i-1][j-w_i]$.

Оскільки нам потрібен максимум, то фінальне рекурентне співвідношення виглядає так: $dp[i][j] = \max(dp[i-1][j], \quad v_i + dp[i-1][j-w_i])$

Псевдокод (табуляція):

```
// Вхід: weights[], values[], W (макс. вага), n (кількість предметів)
ФУНКЦІЯ knapsack_dp(weights, values, W, n):
    // Створюємо ДП-таблицю розміром (n+1) x (W+1)
    dp = нова_матриця(n + 1, W + 1, заповнена 0)

    // Заповнюємо таблицю знизу-вгору
    ДЛЯ i ВІД 1 ДО n:       // Ітеруємо по предметах
        ДЛЯ j ВІД 1 ДО W:   // Ітеруємо по вазі рюкзака
            // Варіант 1: Не беремо i-й предмет
            value_without_i = dp[i-1][j]

            // Варіант 2: Беремо i-й предмет (якщо він влазить)
            value_with_i = 0
            ЯКЩО weights[i-1] <= j ТО
                value_with_i = values[i-1] + dp[i-1][j - weights[i-1]
]
            КІНЕЦЬ ЯКЩО

            // Обираємо кращий з двох варіантів
            dp[i][j] = max(value_without_i, value_with_i)
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ДЛЯ

    // Результат - в правому нижньому куті таблиці
    ПОВЕРНУТИ dp[n][W]
КІНЕЦЬ ФУНКЦІЇ
```

Відновлення набору предметів. Наведена функція знаходить максимальну вартість, але не самі предмети. Щоб їх знайти, потрібно пройти по заповненій таблиці `dp` у зворотному напрямку, від `dp[n][W]`, і аналізувати,

на якому кроці значення змінювалося, що вказуватиме на включення предмета до розв'язку.

2. Область застосування

Динамічне програмування — надзвичайно потужна техніка для розв'язання широкого класу оптимізаційних задач.

- Біоінформатика: Вирівнювання послідовностей ДНК (алгоритм Нідлмана-Вунша базується на ДП).
- Теорія графів: Пошук найкоротших шляхів між усіма парами вершин (алгоритм Флойда-Воршелла).
- Обробка природної мови (NLP): Алгоритм Вітербі для прихованих марковських моделей.
- Економіка та фінанси: Оптимізація інвестиційних портфелів.
- Планування та теорія ігор.

3. Приклади застосування

- Системи перевірки правопису: Алгоритми, що обчислюють редакційну відстань (відстань Левенштейна) між двома словами, щоб запропонувати виправлення, використовують ДП.
- Системи порівняння файлів: Утиліти типу **`diff`** використовують алгоритми на основі ДП для пошуку найдовшої спільної підпослідовності (LCS) між двома файлами.
- Прокладання маршрутів: Деякі алгоритми маршрутизації в комп'ютерних мережах використовують принципи ДП.

Переваги та недоліки

Переваги:

- Оптимальність: Динамічне програмування завжди гарантує знаходження глобально оптимального розв'язку.
- Широкий клас задач: Дозволяє розв'язувати багато задач, які не піддаються жадібним чи іншим простішим підходам.

Недоліки:

- Складність за часом та пам'яттю: Часто вимагає значних обсягів пам'яті для зберігання таблиці розв'язків ($O(n \cdot W)$ для рюкзака) та часу на її заповнення. Не підходить для задач з дуже великим простором станів.

– Нетривіальність розробки: Найскладніший етап — правильно визначити підзадачі та сформулювати рекурентне співвідношення. Це вимагає глибокого аналізу та практики.

Контрольні запитання

3. Опишіть своїми словами, що таке «перекриття підзадач» та «оптимальна підструктура».
4. У чому полягає ключова відмінність між підходами мемоізації та табуляції? Які переваги та недоліки кожного?
5. Чому жадібний підхід (наприклад, брати предмети з найбільшим співвідношенням вартість/вага) не працює для задачі про рюкзак 0/1? Наведіть контрприклад.
6. Поясніть рекурентне співвідношення для задачі про рюкзак 0/1. Що представляють індекси `i` та `j` у таблиці `dp[i][j]`?
7. Наведіть приклад задачі (окрім розглянутих), яка, на вашу думку, може бути розв'язана за допомогою ДП. Обґрунтуйте, чому вона має необхідні властивості.

Методичні вказівки до лабораторної роботи

1. Назва

Розв'язання задачі про рюкзак 0/1 методом динамічного програмування.

2. Мета

Набути практичних навичок у формулюванні рекурентного співвідношення та реалізації ітеративного розв'язку (методом табуляції) для класичної задачі динамічного програмування.

3. Завдання

Створити програму, яка знаходить максимальну вартість предметів, що можна помістити у рюкзак з обмеженою вантажопідйомністю, а також виводить перелік цих предметів.

4. Опис

Програма має приймати на вхід масиви ваг та вартостей предметів, а також максимальну вагу рюкзака. Основна логіка має бути реалізована через створення та заповнення двовимірної ДП-таблиці.

5. Порядок виконання

1. Організуйте вхідні дані: Створіть масиви для ваг та вартостей предметів.
2. Ініціалізуйте ДП-таблицю: Створіть двовимірний масив `dp` розміром `(n+1) x (W+1)` та заповніть його нулями.
3. Заповніть таблицю: Реалізуйте два вкладені цикли для ітерації по предметах та вазі рюкзака, як це описано у псевдокоді. Всередині циклів обчислюйте `dp[i][j]` на основі рекурентного співвідношення.
4. Реалізуйте відновлення відповіді: Після заповнення таблиці напишіть окремий блок коду, який проходить по ній у зворотному напрямку (від `dp[n][W]`), щоб визначити, які предмети були включені до оптимального набору.
5. Виведіть результат: Напишіть функцію `main`, яка запускає алгоритм та виводить на екран максимальну вартість та список обраних предметів.

6. Результат

Додаток, що коректно розв'язує задачу про рюкзак 0/1 та виводить як максимальну вартість, так і сам набір предметів.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко розділена логіка заповнення ДП-таблиці та логіка відновлення розв'язку.
2. Продемонструвати роботу додатку на кількох тестових наборах, включаючи той, де жадібний алгоритм дав би неправильну відповідь.
3. Бути готовим відповісти на питання:
    - Пояснити сенс будь-якої клітинки `dp[i][j]` у вашій таблиці.
    - Яка часова та просторова складність вашого розв'язку?
    - Як працює ваш алгоритм відновлення набору предметів?

Тести

1. Яка з властивостей НЕ є обов'язковою для застосування динамічного програмування?

gggg) Оптимальна підструктура.

hhhh) Перекриття підзадач.

iiii) Властивість жадібного вибору.

jjjj) Всі відповіді a) та b) є обов'язковими.

2. Мемоізація (Top-down) — це…

kkkk) Ітеративний підхід із заповненням таблиці.

llll) Рекурсивний підхід зі збереженням вже обчислених результатів.

mmmm) Жадібний підхід.

nnnn) Метод «Розділяй і володарюй».

3. Яка часова складність розв'язку задачі про рюкзак 0/1 методом ДП з n предметів та місткістю W?

oooo) $O(n + W)$

pppp) $O(n\log n)$

qqqq) $O(n \cdot W)$

rrrr) $O(2^n)$

4. В чому основна відмінність ДП від жадібних алгоритмів?

ssss) ДП завжди швидше.

tttt) ДП розглядає кілька варіантів на кожному кроці, а жадібний — лише один.

uuuu) Жадібні алгоритми використовують рекурсію, а ДП — ні.

vvvv) ДП гарантує оптимальність, а жадібні алгоритми — ні.

5. Табуляція (Bottom-up) зазвичай ефективніша за мемоізацію, тому що:

wwww) Вона використовує менше пам'яті.

xxxx) Вона завжди знаходить кращий розв'язок.

yyyy) Вона не має накладних витрат на рекурсивні виклики функцій.

zzzz) Вона простіша в реалізації.

Відповіді: 1-c, 2-b, 3-c, 4-b, 5-c

Завдання для самостійної роботи

1. Фібоначчі (оптимізація пам'яті). Напишіть ітеративну функцію для обчислення n-го числа Фібоначчі, яка використовує лише $O(1)$ додаткової пам'яті.
2. Найдовша спільна підпослідовність (LCS). Реалізуйте алгоритм на основі ДП для знаходження довжини найдовшої спільної підпослідовкості для двох заданих рядків.

3. Відновлення LCS. Модифікуйте попередній алгоритм так, щоб він виводив не тільки довжину, але й саму найдовшу спільну підпослідовність.
4. Редакційна відстань (відстань Левенштейна). Напишіть функцію, що обчислює мінімальну кількість операцій (вставка, видалення, заміна), необхідних для перетворення одного рядка в інший.
5. Найдовша зростаюча підпослідовність (LIS). Реалізуйте алгоритм на основі ДП для знаходження довжини найдовшої зростаючої підпослідовності у заданому масиві чисел.
6. Оптимізація LIS. Дослідіть та реалізуйте алгоритм для знаходження LIS зі складністю $O(n\log n)$.
7. Задача про сходинки. Ви можете підніматися на сходинки, переступаючи через 1 або 2 сходинки за раз. Скількома способами можна дістатися до n-ої сходинки? Розв'яжіть за допомогою ДП.
8. Задача про здачу (Coin Change). Є необмежена кількість монет певних номіналів. Знайдіть мінімальну кількість монет, необхідну для того, щоб набрати задану суму.
9. Множення матричних ланцюгів. Реалізуйте алгоритм, що знаходить оптимальний порядок множення послідовності матриць для мінімізації кількості скалярних множень.
10. Задача про розрізання стрижня (Rod Cutting). Є стрижень довжиною n та прайс-лист для стрижнів різної довжини. Знайдіть такий спосіб розрізати стрижень, щоб сумарна вартість отриманих частин була максимальною.
11. Задача про підмножину з заданою сумою (Subset Sum). Для заданої множини цілих чисел визначте, чи існує непорожня підмножина, сума елементів якої дорівнює заданому числу S.
12. Задача про розбиття на рівні підмножини (Partition Problem). Визначте, чи можна задану множину чисел розбити на дві підмножини з однаковою сумою.
13. Максимальна вартість шляху в матриці. У матриці, заповненій числами, знайдіть шлях з лівого верхнього кута в правий нижній, рухаючись лише вправо та вниз, з максимальною сумою елементів.

14. Задача про шахи. Скількома способами шахова тура може дістатися з лівої верхньої клітинки дошки m x n до правої нижньої, рухаючись лише вправо та вниз?

15. Есе-порівняння. Напишіть есе на тему «Де проходить межа між жадібними алгоритмами та динамічним програмуванням?». Наведіть приклади задач, де обидва підходи можуть бути розглянуті, та поясніть, чому один з них виявляється кращим.

Динамічне програмування в задачі «Найдовша спільна підпослідовність»

Розглянемо ще один канонічний приклад динамічного програмування. Ця задача демонструє, як ДП може ефективно вирішувати проблеми порівняння та пошуку подібностей у послідовностях, що є фундаментальним завданням у багатьох галузях науки та техніки.

1. Теоретичні відомості

1.1. Постановка задачі: що таке підпослідовність?

Перш ніж розв'язувати задачу, давайте чітко визначимо основні поняття. Уявімо послідовність (наприклад, рядок символів) `S = "ABCDEFG"`.

- Підрядок: Це неперервна частина початкового рядка. Наприклад, `"BCD"` є підрядком `S`.
- Підпослідовність: Це послідовність, яку можна отримати з початкової шляхом видалення нуля або більше символів, не змінюючи порядок решти символів. Наприклад, `"ACEG"` є підпослідовністю `S`, оскільки ми видалили символи B, D, F. А от `"AGC"` — вже ні, бо порядок C і G було змінено.

Тепер ми готові сформулювати задачу про найдовшу спільну підпослідовність (LCS): Маючи дві послідовності, X та Y, потрібно знайти найдовшу можливу підпослідовність, яка є підпослідовністю одночасно і для X, і для Y.

Приклад:

- X = «ABCBDAB»
- Y = «BDCABA»
- Одна зі спільних підпослідовностей — «BAB» (довжина 3).
- Найдовша спільна підпослідовність (LCS) — «BCBA» (довжина 4).

Перебрати всі можливі підпослідовності для обох рядків і порівняти їх — це експоненційно складна задача. На щастя, вона має властивості, що дозволяють застосувати динамічне програмування.

1.2. Пошук оптимальної підструктури

Давайте спробуємо розбити задачу на менші підзадачі. Нехай у нас є дві послідовності: `X` довжиною `m` та `Y` довжиною `n`. Будемо позначати префікс послідовності `X` довжиною `i` як X_i, а префікс `Y` довжиною `j` як Y_j. Наше завдання — знайти довжину LCS для X_m та Y_n.

Розглянемо останні символи наших префіксів: $X[i]$ та $Y[j]$. Можливі два випадки:

Випадок 1: Останні символи збігаються ($X[i] == Y[j]$) Якщо останні символи однакові, то вони точно є частиною найдовшої спільної підпослідовності для префіксів X_i та Y_j. Чому? Бо якби ми їх не включили, ми б точно не змогли отримати довшу підпослідовність. Отже, ми беремо цей спільний символ і тепер нам залишається розв'язати меншу підзадачу: знайти LCS для решти рядків, X_{i-1} та Y_{j-1}. Таким чином, довжина LCS для X_i та Y_j дорівнює 1 + (довжина LCS для X_{i-1} та Y_{j-1}).

Випадок 2: Останні символи не збігаються ($X[i] \neq Y[j]$) У цьому випадку останній символ LCS точно не може бути одночасно і $X[i]$, і $Y[j]$. Отже, нам потрібно зробити вибір. Оптимальний розв'язок для X_i та Y_j буде отримано з одного з двох варіантів:

- Або з оптимального розв'язку для X_{i-1} та Y_j (тобто ми «відкидаємо» символ $X[i]$ і шукаємо LCS для решти).
- Або з оптимального розв'язку для X_i та Y_{j-1} (тобто ми «відкидаємо» символ $Y[j]$).

Оскільки ми шукаємо найдовшу підпослідовність, ми маємо обрати кращий (більший) з цих двох варіантів. Таким чином, довжина LCS для X_i та Y_j дорівнює max(довжина LCS для X_{i-1} та Y_j, довжина LCS для X_i та Y_{j-1}).

1.3. Рекурентне співвідношення та ДП-таблиця

Ми щойно сформулювали властивість оптимальної підструктури. Оскільки при рекурсивному розв'язанні ми будемо багато разів обчислювати LCS для одних і тих самих префіксів, задача також має властивість перекриття підзадач. Це ідеальні умови для ДП.

Давайте використаємо підхід табуляції (bottom-up). Створимо двовимірну таблицю (матрицю) `dp` розміром `(m+1) x (n+1)`. Кожна клітинка `dp[i][j]` буде зберігати довжину LCS для префіксів X_i та Y_j.

Формальне рекурентне співвідношення:

$$dp[i][j] = \begin{cases} 0 & \text{якщо } i = 0 \text{ або } j = 0 \\ 1 + dp[i-1][j-1] & \text{якщо } X[i-1] == Y[j-1] \\ \max(dp[i- \end{cases}$$

Примітка: ми використовуємо індекси `i-1` та `j-1` для доступу до рядків, оскільки рядки зазвичай індексуються з 0, а наша ДП-таблиця має розмір `(m+1)x(n+1)` для зручності обробки базових випадків (пустих префіксів).

Псевдокод для заповнення таблиці:

```
// Вхід: рядки X, Y довжиною m та n відповідно
ФУНКЦІЯ lcs_length(X, Y):
    m = довжина(X)
    n = довжина(Y)

    // Створюємо ДП-таблицю
    dp = нова_матриця(m + 1, n + 1, заповнена 0)

    // Заповнюємо таблицю знизу-вгору
    ДЛЯ i ВІД 1 ДО m:
        ДЛЯ j ВІД 1 ДО n:
            // Випадок 1: символи збігаються
            ЯКЩО X[i-1] == Y[j-1] ТО
                dp[i][j] = 1 + dp[i-1][j-1]
            // Випадок 2: символи не збігаються
            ІНАКШЕ
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
            КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ДЛЯ

    // Результат знаходиться в правому нижньому куті
    ПОВЕРНУТИ dp[m][n]
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Відновлення підпослідовності

Таблиця `dp` дає нам лише довжину LCS. Щоб відновити саму послідовність, потрібно пройти по таблиці у зворотному напрямку, від клітинки `dp[m][n]` до `dp[0][0]`.

Алгоритм відновлення:

2. Починаємо з `i = m`, `j = n`.
3. Якщо $X[i-1] == Y[j-1]$, цей символ є частиною LCS. Додаємо його до нашого результату і переміщуємось по діагоналі вгору-вліво (до `i-1, j-1`).
4. Якщо $X[i-1] \neq Y[j-1]$, ми дивимось на сусідні клітинки: `dp[i-1][j]` та `dp[i][j-1]`. Ми переміщуємось у ту, де значення більше. Якщо вони однакові, можна обрати будь-яку (зазвичай, рухаються вгору).
5. Повторюємо кроки 2-3, доки не досягнемо `i=0` або `j=0`.
6. Оскільки ми збирали символи з кінця, отриманий рядок потрібно перевернути.

Псевдокод для відновлення LCS:

```
ФУНКЦІЯ reconstruct_lcs(X, Y, dp):
    i = довжина(X)
    j = довжина(Y)
    lcs_string = ""

    ПОКИ i > 0 І j > 0:
        // Якщо символи збігаються, вони є частиною LCS
        ЯКЩО X[i-1] == Y[j-1] ТО
            lcs_string = X[i-1] + lcs_string // Додаємо символ на поч
аток
            i = i - 1
            j = j - 1
        // Інакше рухаємось у напрямку більшого значення
        ІНАКШЕ ЯКЩО dp[i-1][j] > dp[i][j-1] ТО
            i = i - 1
        ІНАКШЕ
            j = j - 1
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ПОКИ

    ПОВЕРНУТИ lcs_string
КІНЕЦЬ ФУНКЦІЇ
```

Часова та просторова складність обох етапів (заповнення таблиці та відновлення) становить $O(m \cdot n)$.

2. Область застосування

Алгоритм LCS та його варіації є надзвичайно важливими і застосовуються в багатьох галузях.

- Біоінформатика: Порівняння послідовностей ДНК, РНК та білків для виявлення генетичної подібності, мутацій та еволюційних зв'язків. Алгоритм Нідлмана-Вунша для глобального вирівнювання послідовностей є прямою реалізацією LCS.
- Розробка програмного забезпечення: Системи контролю версій (Git, SVN) використовують алгоритми, подібні до LCS, для реалізації утиліти `diff`, яка показує відмінності між двома версіями файлу.
- Обробка текстів: Виявлення плагіату, порівняння документів.

3. Приклади застосування

- `git diff`: Коли ви виконуєте цю команду, система знаходить найдовшу спільну підпослідовність рядків у двох файлах. Рядки, що не увійшли до LCS, позначаються як додані або видалені.
- BLAST (Basic Local Alignment Search Tool): Інструмент у біоінформатиці, що швидко знаходить регіони локальної подібності між біологічними послідовностями, використовуючи евристичні версії алгоритмів вирівнювання.

Переваги та недоліки

Переваги:

- Гарантована оптимальність: Динамічне програмування завжди знаходить найдовшу спільну підпослідовність.
- Універсальність: Підхід легко адаптується до схожих задач (напр., редакційна відстань).
- Простота концепції: Після розуміння рекурентного співвідношення, реалізація методом табуляції є досить прямолінійною.

Недоліки:

- Квадратична складність: Часова та просторова складність $O(m \cdot n)$ може бути занадто великою для дуже довгих послідовностей (наприклад, для повних геномів).
- Вимоги до пам'яті: Зберігання повної ДП-таблиці може бути проблематичним. Існують оптимізації (напр., алгоритм Гіршберга), що дозволяють зменшити вимоги до пам'яті до $O(\min(m, n))$, але вони складніші в реалізації.

Контрольні запитання

1. У чому полягає принципова відмінність між підпослідовністю та підрядком? Наведіть приклад.
2. Поясніть логіку рекурентного співвідношення для LCS: чому ми розглядаємо два випадки (символи збігаються / не збігаються)?
3. Що зберігається в кожній клітинці `dp[i][j]` ДП-таблиці?
4. Опишіть, як працює алгоритм відновлення відповіді. Чому ми рухаємось від правого нижнього кута таблиці до лівого верхнього?
5. Наведіть приклад задачі з реального життя, де можна було б застосувати алгоритм LCS.

Методичні вказівки до лабораторної роботи

1. Назва

Знаходження найдовшої спільної підпослідовності методом динамічного програмування.

2. Мета

Практичне засвоєння методу динамічного програмування для розв'язання задач на послідовностях. Реалізація алгоритму заповнення ДП-таблиці та алгоритму зворотного проходу для відновлення розв'язку.

3. Завдання

Створити програму, яка для двох заданих рядків знаходить:

1. Довжину їхньої найдовшої спільної підпослідовності.
2. Саму найдовшу спільну підпослідовність.

4. Опис

Програма має приймати на вхід два рядки. Основна логіка має бути реалізована через створення та заповнення двовимірної ДП-таблиці, а потім виконання зворотного проходу по ній для реконструкції LCS.

5. Порядок виконання

1. Отримайте вхідні дані: Напишіть код, що зчитує два рядки X та Y.
2. Ініціалізуйте ДП-таблицю: Створіть двовимірний масив `dp` розміром `(довжина(X)+1) x (довжина(Y)+1)` та заповніть його нулями.
3. Заповніть таблицю: Керуючись псевдокодом, реалізуйте два вкладені цикли для заповнення таблиці `dp` на основі рекурентного співвідношення.

4. Виведіть довжину: Після заповнення таблиці, значення `dp[m][n]` і буде довжиною LCS. Виведіть його.
5. Реалізуйте відновлення LCS: Напишіть функцію, що реалізує алгоритм зворотного проходу по таблиці `dp`, як це описано у псевдокоді.
6. Виведіть результат: Викличте функцію відновлення та виведіть на екран саму найдовшу спільну підпослідовність.

6. Результат

Додаток, що для будь-якої пари вхідних рядків коректно знаходить і виводить довжину LCS та один із можливих варіантів самої LCS.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з чіткою реалізацією обох етапів: заповнення таблиці та відновлення розв'язку.
2. Продемонструвати роботу додатку на кількох тестових парах рядків (включаючи ті, що не мають спільних символів, та ті, де один рядок є підпослідовністю іншого).
3. Бути готовим відповісти на питання:
    - Пояснити рекурентне співвідношення, яке ви реалізували.
    - Яка часова та просторова складність вашої програми?
    - Як зміниться результат, якщо в алгоритмі відновлення при незбігу символів і рівних значеннях у сусідніх клітинках рухатись не вгору, а вліво?

Тести

1. Для рядків X=«AGGTAB» та Y=«GXTXAYB», яка з послідовностей є їхньою спільною підпослідовністю?

aaaaa) «GTAB»

bbbbb) «GXT»

ccccc) «AGB»

ddddd) «GTB»

2. Яка часова складність алгоритму знаходження довжини LCS для рядків довжиною m та n?

eeeee) $O(m + n)$

fffff) $O(m \cdot n)$

ggggg) $O(m \log n)$

hhhhh) $O(2^n)$

3. Якщо при відновленні LCS з таблиці `dp` ми знаходимось в клітинці `(i, j)` і `X[i-1] == Y[j-1]`, яким буде наш наступний крок?

iiiii) Перемістити до `dp[i-1][j]`.

jjjjj) Перемістити до `dp[i][j-1]`.

kkkkk) Перемістити до `dp[i-1][j-1]`.

lllll) Завершити алгоритм.

4. Алгоритм LCS є класичним прикладом:

mmmmm) Жадібних алгоритмів.

nnnnn) Методу «Розділяй і володарюй».

ooooo) Пошуку з поверненням.

ppppp) Динамічного програмування.

5. Яка головна перевага ДП-рішення для LCS над повним перебором?

qqqqq) Використовує менше пам’яті.

rrrrr) Гарантує знаходження коротшої підпослідовності.

sssss) Уникає повторного обчислення розв’язків для однакових підзадач.

ttttt) Простіший в реалізації.

Відповіді: 1-d, 2-b, 3-c, 4-d, 5-c

Завдання для самостійної роботи

1. Мемоізація. Реалізуйте знаходження довжини LCS за допомогою рекурсивного підходу з мемоізацією (top-down).
2. Найкоротша спільна надпослідовність (Shortest Common Supersequence). Напишіть програму, яка для двох рядків знаходить найкоротший рядок, що містить обидва початкові рядки як свої підпослідовності. (Підказка: це тісно пов’язано з довжиною LCS).
3. Найдовша паліндромна підпослідовність. Для заданого рядка знайдіть найдовшу підпослідовність, яка є паліндромом. (Підказка: LCS рядка та його перевернутої версії).
4. Редакційна відстань (Відстань Левенштейна). Реалізуйте алгоритм, що знаходить мінімальну кількість операцій (вставка, видалення, заміна)

для перетворення одного рядка в інший. Порівняйте його рекурентне співвідношення з LCS.

5. Оптимізація пам’яті. Реалізуйте алгоритм знаходження лише довжини LCS, використовуючи тільки $O(\min(m,n))$ додаткової пам’яті. (Підказка: для обчислення поточного рядка ДП-таблиці потрібен лише попередній).
6. Алгоритм Гіршберга. Дослідіть та опишіть алгоритм Гіршберга, який дозволяє знайти саму LCS (не тільки довжину), використовуючи $O(\min(m,n))$ пам’яті.
7. LCS для трьох рядків. Узагальніть алгоритм LCS для знаходження найдовшої спільної підпослідовності для трьох заданих рядків. Якою буде складність такого алгоритму?
8. Найдовша підпослідовність з повтореннями (Longest Repeated Subsequence). Знайдіть найдовшу підпослідовність, яка зустрічається в рядку щонайменше двічі (при цьому входження не повинні перетинатися за індексами). (Підказка: LCS рядка з самим собою, з додатковою умовою).
9. Кількість різних LCS. Модифікуйте алгоритм так, щоб він підраховував не довжину, а кількість унікальних найдовших спільних підпослідовностей.
10. Найважча спільна підпослідовність. Уявіть, що кожен символ алфавіту має свою «вагу». Модифікуйте алгоритм LCS так, щоб він знаходив спільну підпослідовність з максимальною сумарною вагою.
11. Найдовший спільний підрядок. Напишіть алгоритм на основі ДП для знаходження найдовшого спільного підрядка (неперервної частини) двох рядків.
12. «Дика карта». Адаптуйте алгоритм LCS для випадку, коли один з рядків може містити спеціальний символ ‘?’ (дика карта), який може збігатися з будь-яким символом.
13. LCS з обмеженнями. Знайдіть LCS двох рядків з додатковим обмеженням, що вона не повинна містити певний заборонений підрядок.
14. Візуалізація ДП-таблиці. Напишіть програму, яка не тільки обчислює LCS, але й виводить на екран заповнену ДП-таблицю та візуалізує шлях відновлення відповіді.

15. Есе-аналіз. Напишіть есе на тему «Застосування LCS-подібних алгоритмів поза біоінформатикою та порівнянням файлів». Знайдіть та опишіть 2-3 неочевидні сфери застосування.

Динамічне програмування в задачі «Редакційна відстань»

Після того, як ми навчилися знаходити спільне у послідовностях, розглянемо споріднену, але іншу за метою задачу: як виміряти «відмінність» між ними. Цей алгоритм є основою для багатьох систем, з якими ми стикаємось щодня, від перевірки орфографії до біоінформатики.

1. Теоретичні відомості

1.1. Постановка задачі: вимірювання «схожості» рядків

Уявіть, що ви набираєте слово «алгоритм», але випадково друкуєте «алогоритм». Система перевірки орфографії миттєво пропонує вам правильний варіант. Як вона це робить? Вона не знає, що ви мали на увазі, але вона може обчислити, що «алогоритм» «ближче» до «алгоритм», ніж до будь-якого іншого слова у словнику.

Щоб кількісно виміряти цю «близькість», використовується поняття редакційної відстані. Це мінімальна кількість базових операцій, необхідних для перетворення одного рядка в інший. Найбільш поширеною є відстань Левенштейна, яка враховує три операції:

- Вставка (Insertion): додати один символ.
- Видалення (Deletion): видалити один символ.
- Заміна (Substitution): замінити один символ на інший.

Отже, відстань Левенштейна між двома рядками — це мінімальна кількість вставок, видалень та замін, щоб перетворити один рядок на інший. Наприклад, відстань між «cat» і «car» дорівнює 1 (одна заміна). Відстань між «sunday» і «saturday» дорівнює 3 (вставка ‘a’ і ‘t’, заміна ‘n’ на ‘r’).

1.2. Пошук оптимальної підструктури та рекурентне співвідношення

Як і у випадку з LCS, ця задача має оптимальну підструктуру. Розв’язок для двох довгих рядків можна побудувати з розв’язків для їхніх коротших префіксів.

Нехай **`dp[i][j]`** — це відстань Левенштейна між першими **`i`** символами рядка X (префікс X_i) та першими **`j`** символами рядка Y (префікс Y_j). Ми будемо будувати ДП-таблицю розміром **`(m+1) x (n+1)`**.

Базові випадки:

– Відстань між рядком довжиною i та порожнім рядком — це i операцій видалення. Отже, $dp[i][0] = i$.

– Відстань між порожнім рядком та рядком довжиною j — це j операцій вставки. Отже, $dp[0][j] = j$. Це дає нам заповнені перший рядок та перший стовпець нашої таблиці.

Рекурсивний крок: Тепер розглянемо, як обчислити `dp[i][j]`, знаючи попередні значення. Ми дивимось на останні символи наших префіксів: $X[i-1]$ та $Y[j-1]$.

– Випадок 1: Останні символи збігаються ($X[i-1] == Y[j-1]$). Якщо символи однакові, то для них не потрібно жодної операції. Редакційна відстань між X_i та Y_j така сама, як і відстань між префіксами без цих останніх символів, тобто X_{i-1} та Y_{j-1}. $dp[i][j] = dp[i-1][j-1]$

– Випадок 2: Останні символи не збігаються ($X[i-1] \neq Y[j-1]$). Тут ми маємо виконати одну з трьох можливих операцій. Ми обираємо ту, яка є «найдешевшою» (веде до мінімальної загальної відстані).

  1. Вставка: Ми можемо перетворити X_i на Y_j, якщо спершу перетворимо X_i на Y_{j-1} (вартість `dp[i][j-1]`), а потім вставимо символ $Y[j-1]$. Загальна вартість: $1 + dp[i][j-1]$.
  2. Видалення: Ми можемо перетворити X_i на Y_j, якщо спершу перетворимо X_{i-1} на Y_j (вартість `dp[i-1][j]`), а потім видалимо символ $X[i-1]$. Загальна вартість: $1 + dp[i-1][j]$.
  3. Заміна: Ми можемо перетворити X_i на Y_j, якщо спершу перетворимо X_{i-1} на Y_{j-1} (вартість `dp[i-1][j-1]`), а потім замінимо символ $X[i-1]$ на $Y[j-1]$. Загальна вартість: $1 + dp[i-1][j-1]$.

  Ми обираємо мінімум з цих трьох варіантів: $dp[i][j] = 1 + \min(dp[i-1][j],\quad dp[i][j-1],\quad dp[i-1][j-1])$

1.3. Псевдокод та візуалізація таблиці

Псевдокод для обчислення відстані Левенштейна:

```
// Вхід: рядки X, Y довжиною m та n відповідно
ФУНКЦІЯ LevenshteinDistance(X, Y):
    m = довжина(X)
    n = довжина(Y)

    // Створюємо ДП-таблицю
    dp = нова_матриця(m + 1, n + 1)
```

```
    // Крок 1: Ініціалізуємо базові випадки
    ДЛЯ i ВІД 0 ДО m:
        dp[i][0] = i
    КІНЕЦЬ ДЛЯ
    ДЛЯ j ВІД 0 ДО n:
        dp[0][j] = j
    КІНЕЦЬ ДЛЯ

    // Крок 2: Заповнюємо решту таблиці
    ДЛЯ i ВІД 1 ДО m:
        ДЛЯ j ВІД 1 ДО n:
            // Вартість заміни: 0, якщо символи однакові, 1 - якщо рі
зні
            cost = (ЯКЩО X[i-1] == Y[j-1] ТО 0 ІНАКШЕ 1)

            // Обчислюємо значення клітинки як мінімум з трьох сусідн
іх + вартість
            dp[i][j] = min(dp[i-1][j] + 1,        // Видалення
                           dp[i][j-1] + 1,        // Вставка
                           dp[i-1][j-1] + cost)   // Заміна або збіг
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ДЛЯ

    // Результат - в правому нижньому куті таблиці
    ПОВЕРНУТИ dp[m][n]
КІНЕЦЬ ФУНКЦІЇ
```

Візуалізація. Уявімо, що ми перетворюємо «SATURDAY» на «SUNDAY». ДП-таблиця покаже нам мінімальну вартість перетворення будь-якого префікса першого слова на будь-який префікс другого. Клітинка `dp[8][6]` буде містити фінальну відстань.

1.4. Відновлення редакційних операцій

Як і у випадку з LCS, таблиця дає нам лише число — відстань. Щоб дізнатися, які саме операції потрібно виконати, ми маємо пройти по таблиці у зворотному напрямку, від `dp[m][n]`.

Алгоритм відновлення:

1. Починаємо з `i = m`, `j = n`.
2. На кожному кроці дивимось на поточний символ `cost` (0 або 1) та значення трьох сусідніх клітинок: `dp[i-1][j]`, `dp[i][j-1]`, `dp[i-1][j-1]`.

3. Обираємо ту клітинку, яка дала мінімальне значення для `dp[i][j]`, і рухаємось до неї.
    - Якщо ми перейшли з `dp[i-1][j-1]` і `cost` був 0, це означає «збіг, без операцій».
    - Якщо ми перейшли з `dp[i-1][j-1]` і `cost` був 1, це означає «заміна».
    - Якщо ми перейшли з `dp[i-1][j]`, це означає «видалення» символу $X[i-1]$.
    - Якщо ми перейшли з `dp[i][j-1]`, це означає «вставка» символу $Y[j-1]$.
4. Повторюємо, доки не дійдемо до `dp[0][0]`.

2. Область застосування

Редакційна відстань є фундаментальним поняттям в обробці текстової інформації.

– Перевірка орфографії (Spell Checking): Пропозиція виправлень для слів з помилками на основі найменшої редакційної відстані до слів зі словника.

– Біоінформатика: Порівняння послідовностей ДНК, де помилки можуть виникати через мутації (вставки, видалення, заміни нуклеотидів).

– Обробка природної мови (NLP): Оцінка помилок у системах розпізнавання мови (ASR) та машинного перекладу.

– Пошукові системи: Реалізація функції «Можливо, ви мали на увазі…?» для нечіткого пошуку.

3. Приклади застосування

– Редактори коду (VS Code, Sublime Text): Коли ви шукаєте щось у файлі, багато редакторів використовують «нечіткий пошук» (fuzzy search), який знаходить збіги навіть з невеликими помилками, базуючись на редакційній відстані.

– Системи виявлення плагіату: Хоча використовуються більш складні алгоритми, в їх основі лежить ідея порівняння фрагментів тексту на «схожість», яка може бути виміряна редакційною відстанню.

Переваги та недоліки

Переваги:

- Гарантована оптимальність: ДП завжди знаходить мінімальну кількість операцій.
- Універсальність та гнучкість: Алгоритм легко модифікувати, наприклад, призначивши різні «штрафи» за різні операції (вага заміни може бути 2, а вставки/видалення — 1).

Недоліки:

- Квадратична складність: Часова та просторова складність $O(m \cdot n)$ є головним обмежуючим фактором для дуже довгих рядків.
- Не враховує семантику: Для алгоритму слова «дім» і «діл» однаково далекі від слова «дим» (відстань 1). Він не враховує фонетичної чи семантичної близькості.

Контрольні запитання

1. Дайте визначення відстані Левенштейна. Які три операції вона враховує?
2. Поясніть, як ініціалізується перший рядок та перший стовпець ДП-таблиці для відстані Левенштейна і чому саме так.
3. Запишіть та поясніть рекурентне співвідношення для випадку, коли останні символи префіксів не збігаються.
4. У чому полягає ключова відмінність у рекурентному співвідношенні для LCS та відстані Левенштейна?
5. Як можна модифікувати алгоритм, щоб вартість заміни символу була 2, а вартість вставки/видалення — 1?

Методичні вказівки до лабораторної роботи

1. Назва

Обчислення відстані Левенштейна методом динамічного програмування.

2. Мета

Набути практичних навичок у розв’язанні оптимізаційних задач на рядках методом динамічного програмування, реалізувавши алгоритм обчислення редакційної відстані.

3. Завдання

Створити програму, яка для двох заданих рядків обчислює відстань Левенштейна між ними.

4. Опис

Програма має приймати на вхід два рядки. Основна логіка має бути реалізована через створення та заповнення двовимірної ДП-таблиці відповідно до рекурентного співвідношення.

5. Порядок виконання

1. Отримайте вхідні дані: Напишіть код, що зчитує два рядки X та Y.
2. Ініціалізуйте ДП-таблицю: Створіть двовимірний масив `dp` розміром `(довжина(X)+1) x (довжина(Y)+1)`.
3. Заповніть базові випадки: Ініціалізуйте перший рядок та перший стовпець таблиці значеннями від 0 до n та від 0 до m відповідно.
4. Заповніть решту таблиці: Керуючись псевдокодом, реалізуйте два вкладені цикли для заповнення таблиці `dp`.
5. Виведіть результат: Значення в правому нижньому куті таблиці, `dp[m][n]`, є кінцевою відповіддю. Виведіть його.
6. (Завдання підвищеної складності) Реалізуйте алгоритм зворотного проходу для відновлення послідовності операцій.

6. Результат

Додаток, що для будь-якої пари вхідних рядків коректно обчислює та виводить їхню відстань Левенштейна.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією алгоритму.
2. Продемонструвати роботу додатку на кількох тестових парах рядків (напр., «apple» та «apply»; «kitten» та «sitting»).
3. Бути готовим відповісти на питання:
    - Пояснити рекурентне співвідношення, яке ви реалізували.
    - Яка часова та просторова складність вашої програми?
    - Що зберігає клітинка `dp[i][j]` у вашій реалізації?

Тести

1. Яка відстань Левенштейна між рядками «book» та «back»?

а) 1

b) 2

c) 3

d) 4

3. Якщо при обчисленні `dp[i][j]` символи `X[i-1]` та `Y[j-1]` збігаються, то `dp[i][j]` дорівнює:

a) `dp[i-1][j-1]`

b) `1 + dp[i-1][j-1]`

c) `min(dp[i-1][j], dp[i][j-1])`

d) 0

3. Яка операція НЕ входить до класичного визначення відстані Левенштейна?

a) Вставка

b) Видалення

c) Заміна

d) Транспозиція (перестановка двох сусідніх символів)

5. Базовий випадок `dp[0][j] = j` відповідає перетворенню:

a) Рядка Y в порожній рядок шляхом j видалень.

b) Порожнього рядка в рядок Y шляхом j вставок.

c) Рядка X в рядок Y шляхом j замін.

d) Це помилкова ініціалізація.

5. Відстань Левенштейна є прикладом задачі, що має властивості:

a) Лише оптимальної підструктури.

b) Лише перекриття підзадач.

c) Оптимальної підструктури та перекриття підзадач.

d) Властивості жадібного вибору.

Відповіді: 1-b, 2-a, 3-d, 4-b, 5-c

Завдання для самостійної роботи

1. Відновлення операцій. Модифікуйте програму з лабораторної роботи так, щоб вона виводила не лише відстань, а й саму послідовність операцій (напр., «Замінити ‘o’ на ‘a’ в позиції 1, замінити ‘o’ на ‘c’ в позиції 2»).

2. Оптимізація пам’яті. Реалізуйте алгоритм обчислення відстані Левенштейна, використовуючи лише два рядки (або навіть один) ДП-таблиці, досягнувши просторової складності $O(\min(m,n))$.
3. Відстань Дамерау-Левенштейна. Дослідіть та реалізуйте алгоритм для відстані Дамерау-Левенштейна, яка додатково враховує операцію транспозиції (перестановки двох сусідніх символів, напр., «ab» -> «ba») вартістю 1.
4. Зважена відстань. Модифікуйте алгоритм так, щоб вартість кожної операції була різною (напр., вставка=1, видалення=1, заміна=2).
5. LCS через відстань Левенштейна. Дослідіть та опишіть, як можна знайти довжину LCS двох рядків, використовуючи модифіковану версію алгоритму відстані Левенштейна.
6. Найдовший спільний підрядок. Напишіть алгоритм на основі ДП для знаходження довжини найдовшого спільного підрядка (неперервної частини) двох рядків.
7. Пошук з «дикими картами». Реалізуйте алгоритм порівняння двох рядків, де один з них може містити символ `*`, що відповідає будь-якій послідовності символів (включаючи порожню).
8. Перевірка на відстань. Напишіть функцію `isWithinDistance(X, Y, k)`, яка ефективно перевіряє, чи є відстань Левенштейна між X та Y не більшою за `k`, не обчислюючи всю ДП-таблицю.
9. Найближче слово у словнику. Напишіть програму, яка для заданого слова з помилкою знаходить найближче до нього слово із невеликого словника, використовуючи відстань Левенштейна.
10. Максимальна кількість збігів. Модифікуйте алгоритм, щоб він максимізував кількість збігів, а не мінімізував кількість редагувань.
11. Вирівнювання з розривами (Gap Penalty). Дослідіть алгоритм Нідлмана-Вунша, який використовується в біоінформатиці. Опишіть, як в ньому реалізовано штрафи за початок розриву (gap opening) та продовження розриву (gap extension).
12. Візуалізатор ДП-таблиці. Створіть програму, що візуалізує заповнення ДП-таблиці для відстані Левенштейна та показує шлях відновлення відповіді.
13. Афінний штраф за пропуск. Реалізуйте алгоритм обчислення редакційної відстані з афінним штрафом за пропуск, де перший

пропуск (вставка/видалення) коштує `A+B`, а кожен наступний поспіль — `B`.

14. Бітопаралельний алгоритм. Дослідіть та опишіть ідею бітопаралельного алгоритму (алгоритм Майєрса) для обчислення редакційної відстані.
15. Есе-аналіз. Напишіть есе на тему «Застосування метрик відстані між рядками в сучасному світі: від біоінформатики до боротьби з плагіатом».

Динамічне програмування на підмножинах

Тепер давайте зануримось у спеціалізовану, але надзвичайно потужну техніку — динамічне програмування по підмножинах з використанням бітових масок. Цей метод дозволяє елегантно розв'язувати задачі, де стан залежить від конкретного набору використаних елементів.

1. Теоретичні відомості

1.1. Коли звичайного ДП недостатньо? Вступ до ДП на підмножинах (Bitmask DP)

У попередніх розділах ми розв'язували задачі, де стан ДП можна було описати простими параметрами. Наприклад, у задачі про рюкзак стан був `dp[i][w]`, що означало «максимальна вартість, яку можна отримати, розглянувши перші *i* предметів при вазі *w*». Нам було неважливо, *які саме* предмети ми взяли, а лише *скільки* їх було розглянуто.

Але що робити, якщо розв'язок залежить не від кількості, а від конкретного набору елементів? Уявіть задачу комівояжера: знаючи, що ми відвідали 3 міста, нам критично важливо знати, *які саме* це були 3 міста, щоб планувати подальший шлях. Звичайний підхід тут не спрацює.

Саме для таких задач, де стан залежить від підмножини елементів, і була створена техніка Bitmask DP. Вона дозволяє розв'язувати NP-складні задачі повним перебором по станах, але робить це значно ефективніше за наївний перебір, коли кількість елементів невелика (зазвичай, до 20-22).

1.2. Інструментарій: Бітові маски та операції з ними

Ключова ідея Bitmask DP — кодувати підмножину елементів за допомогою цілого числа, яке ми називаємо бітовою маскою. Уявіть, що у нас є N елементів, пронумерованих від 0 до N-1. Ми можемо представити будь-яку підмножину цих елементів за допомогою N-бітного числа (маски).

Принцип: Якщо *i*-й біт у масці дорівнює 1, це означає, що *i*-й елемент входить до підмножини. Якщо 0 — то не входить.

Приклад: Нехай N=4, елементи {0, 1, 2, 3}.

- Підмножина `{0, 2}` представляється як... `...00101` (двійкове). Це число 5.
- Підмножина `{1, 2, 3}` представляється як... `...01110` (двійкове). Це число 14.
- Повна множина `{0, 1, 2, 3}` — `1111` (двійкове), число 15.
- Порожня множина `{}` — `0000` (двійкове), число 0.

Для роботи з масками ми використовуємо побітові операції:

- Встановити *i*-й біт (додати елемент *i* до множини): `mask = mask | (1 << i)`
  - `1 << i` створює число, де лише *i*-й біт дорівнює 1 (напр., `1 << 2` це `100`). Операція `OR` (|) додає цю одиницю до маски.
- Перевірити, чи встановлений *i*-й біт (чи є елемент *i* у множині): `if (mask & (1 << i)) != 0`
  - Операція `AND` (&) з маскою `...1...` (де 1 на *i*-й позиції) дасть ненульовий результат тільки якщо *i*-й біт у початковій масці теж був 1.
- Очистити *i*-й біт (видалити елемент *i* з множини): `mask = mask & ~(1 << i)`
  - `~(1 << i)` створює маску, де всі біти — одиниці, окрім *i*-го. `AND` з такою маскою «обнуляє» *i*-й біт.
- Переключити *i*-й біт (додати, якщо не було; видалити, якщо був): `mask = mask ^ (1 << i)`
  - Операція `XOR` (^) інвертує біт.

1.3. Детальний розбір: Задача комівояжера (TSP) методом Bitmask DP

Ми вже розглядали цю NP-складну задачу в контексті методу гілок та меж. Давайте розв'яжемо її тепер за допомогою ДП.

Постановка задачі: Є N міст, задані відстані між ними. Потрібно знайти найкоротший маршрут, що починається у місті 0, відвідує кожне інше місто рівно один раз і повертається в місто 0.

Формулювання стану ДП: Нам потрібно зберігати інформацію про те, які міста ми вже відвідали, і в якому місті ми знаходимось зараз. `dp[mask][i]` — це довжина найкоротшого шляху, що:

1. Починається у місті 0.
2. Відвідує рівно ті міста, що вказані одиницями в бітовій масці `mask`.
3. Закінчується у місті `i`.

Розмір таблиці: `mask` може приймати значення від 0 до $2^N - 1$. `i` — від 0 до N-1. Отже, нам потрібна таблиця розміром $2^N \times N$.

Рекурентне співвідношення:

– Базовий випадок: Шлях починається в місті 0 і відвідує лише його.
  - Маска, що представляє множину `{0}`, це `1 << 0` (тобто, 1).
  - `dp[1 << 0][0] = 0`.
  - Всі інші значення `dp` ініціалізуємо нескінченністю.
– Рекурсивний крок: Щоб обчислити `dp[mask][i]` (вартість шляху до міста `i` через множину `mask`), ми маємо прийти в місто `i` з якогось попереднього міста `j`. Це місто `j` також має належати до множини `mask` і, звісно, `j` не може бути `i`.
  - Попередній стан, з якого ми прийшли, це шлях, що закінчується в місті `j` і відвідує всі міста з `mask`, окрім міста `i`. Маска для такого стану — це `prev_mask = mask ^ (1 << i)`.
  - Вартість шляху до попереднього стану — `dp[prev_mask][j]`.
  - Вартість переходу з `j` в `i` — `distance(j, i)`.
  - Ми маємо перебрати всі можливі попередні міста `j` і обрати той шлях, що дає мінімальну вартість.

Отже, фінальне співвідношення: $dp[\text{mask}][i] = \min_{j \in \text{mask}, j \neq i, (j,i)-\text{ребро існує}}(dp[\text{mask} \oplus (1 \ll i)][j] + \text{distance}(j, i))$

1.4. Псевдокод та фінальний результат

```
// Вхід: distance_matrix[N][N], N - кількість міст
ФУНКЦІЯ solve_tsp_dp(distance_matrix, N):
    // Ініціалізуємо ДП-таблицю
    dp = нова_матриця(2^N, N, заповнена +∞)

    // Базовий випадок: шлях починається і закінчується в місті 0
    dp[1 << 0][0] = 0
```

```
    // Ітеруємо по всіх можливих підмножинах (масках)
    ДЛЯ mask ВІД 1 ДО (2^N - 1):
        // Ітеруємо по всіх можливих кінцевих містах 'i' у поточній п
ідмножині
        ДЛЯ i ВІД 0 ДО N-1:
            // Перевіряємо, чи місто 'i' є в поточній підмножині 'mas
k'
            ЯКЩО (mask & (1 << i)) != 0 ТО
                // Ітеруємо по всіх можливих попередніх містах 'j' у
цій же підмножині
                ДЛЯ j ВІД 0 ДО N-1:
                    ЯКЩО i != j І (mask & (1 << j)) != 0 ТО
                        // Визначаємо маску попереднього стану
                        prev_mask = mask ^ (1 << i)
                        // Оновлюємо значення, якщо знайшли коротший
шлях
                        new_dist = dp[prev_mask][j] + distance_matrix
[j][i]
                        dp[mask][i] = min(dp[mask][i], new_dist)
                    КІНЕЦЬ ЯКЩО
                КІНЕЦЬ ДЛЯ
            КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ДЛЯ

    // Шукаємо фінальний результат
    // Це вартість повного шляху + вартість повернення в місто 0
    final_mask = (1 << N) - 1 // Маска, де всі біти = 1 (всі міста ві
двідані)
    min_tour_cost = +∞
    ДЛЯ i ВІД 1 ДО N-1:
        tour_cost = dp[final_mask][i] + distance_matrix[i][0]
        min_tour_cost = min(min_tour_cost, tour_cost)
    КІНЕЦЬ ДЛЯ

    ПОВЕРНУТИ min_tour_cost
КІНЕЦЬ ФУНКЦІЇ
```

Аналіз складності: Зовнішній цикл виконується 2^N разів. Два внутрішні цикли виконуються по N разів. Отже, загальна часова складність — $O(N^2 \cdot 2^N)$. Просторова складність — $O(N \cdot 2^N)$ для зберігання ДП-таблиці. Це значно краще, ніж $O(N)$ для наївного перебору всіх перестановок.

2. Область застосування

Bitmask DP — це специфічна техніка для NP-складних задач на невеликих множинах.

- Задачі на графах: Пошук гамільтонових шляхів/циклів, задача комівояжера.
- Задачі про призначення: Задача про призначення (Assignment Problem), де потрібно розподілити N завдань між N виконавцями з мінімальною вартістю.
- Задачі покриття та пакування: Пошук оптимального способу покрити або упакувати щось, використовуючи елементи з заданого набору.
- Комбінаторні ігри та головоломки.

3. Приклади застосування

- Оптимізація маршрутів: Для невеликої кількості точок (напр., 15-20 пунктів доставки в межах одного району) цей метод може знайти гарантовано оптимальний маршрут, на відміну від евристик.
- Розподіл завдань в багатопроцесорних системах: Коли потрібно оптимально розподілити невелику кількість взаємозалежних завдань між ядрами процесора.
- Спортивне програмування: Це дуже популярна техніка на змаганнях з програмування (ICPC, TopCoder), де часто зустрічаються задачі з обмеженням N ⩽ 20.

Переваги та недоліки

Переваги:

- Точність: Гарантовано знаходить оптимальний розв’язок для NP-складних задач.
- Значно краще за повний перебір: Складність $O(N^2 \cdot 2^N)$ є величезним покращенням порівняно з $O(N)$.

Недоліки:

- Експоненційна складність: Алгоритм все ще має експоненційну складність і стає непридатним для N > 22-24 через обмеження часу та пам’яті.
- Обмежений розмір вхідних даних: Застосовується лише до задач з малою кількістю елементів у множині.

Контрольні запитання

1. Поясніть, як бітова маска може представляти підмножину елементів. Наведіть приклад для N=5.
2. Яка основна відмінність між задачами, що розв'язуються класичним ДП (як рюкзак), і задачами, що вимагають Bitmask DP?
3. Дайте визначення стану `dp[mask][i]` в алгоритмі для задачі комівояжера.
4. Поясніть рекурентне співвідношення для TSP. Чому ми перебираємо всі можливі попередні вершини `j`?
5. Чому часова складність алгоритму Bitmask DP для TSP оцінюється як $O(N^2 \cdot 2^N)$? Поясніть роль кожного множника.

Методичні вказівки до лабораторної роботи

1. Назва

Розв'язання задачі комівояжера (TSP) методом динамічного програмування з бітовими масками.

2. Мета

Набути практичних навичок у застосуванні техніки Bitmask DP для розв'язання NP-складних задач на невеликих наборах даних.

3. Завдання

Створити програму, яка знаходить довжину найкоротшого маршруту комівояжера для заданого зваженого графа, використовуючи Bitmask DP.

4. Опис

Граф задається у вигляді матриці суміжності `distance_matrix`. Основна логіка реалізується через створення та ітеративне заповнення ДП-таблиці `dp[mask][i]`.

5. Порядок виконання

1. Отримайте вхідні дані: Прочитайте матрицю відстаней та кількість міст N.
2. Ініціалізуйте ДП-таблицю: Створіть двовимірний масив `dp` розміром $2^N \times N$ та ініціалізуйте всі його значення нескінченністю.
3. Встановіть базовий випадок: Задайте `dp[1][0] = 0`.

4. Реалізуйте головні цикли: Напишіть цикл по всіх масках від 1 до $2^N - 1$ та вкладені цикли по поточному місту `i` та попередньому `j`, як це описано у псевдокоді.
5. Реалізуйте логіку оновлення: Всередині циклів перевіряйте, чи належать міста масці, та оновлюйте `dp[mask][i]` за рекурентним співвідношенням.
6. Знайдіть фінальний результат: Після заповнення таблиці напишіть окремий цикл, щоб знайти мінімальну вартість повного туру, враховуючи повернення до стартового міста.
7. Протестуйте: Перевірте роботу на невеликому графі (4-5 міст), для якого можна знайти оптимальний розв'язок вручну.

6. Результат

Додаток, що для заданого графа коректно обчислює довжину найкоротшого маршруту комівояжера.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з чіткою реалізацією ДП-таблиці та циклів її заповнення.
2. Продемонструвати роботу додатку на кількох тестових графах.
3. Бути готовим відповісти на питання:
    - Пояснити, як бітові операції використовуються у вашому коді для маніпуляції станами.
    - Яка часова та просторова складність вашої реалізації і чому вона не може працювати для N=30?
    - Як можна модифікувати ваш код для відновлення самого шляху, а не лише його довжини?

Тести

1. Bitmask DP найчастіше застосовується для задач, де N (кількість елементів) є:

a) Дуже великим (N > 1000)

b) Будь-яким

c) Невеликим (N < 25)

d) Лише парним

2. Яка бітова операція використовується, щоб перевірити, чи належить елемент `i` до множини, представленої маскою `mask`?

a) `mask | (1 << i)`

b) `mask & (1 << i)`

c) `mask ^ (1 << i)`

d) `~mask`

3. Яка часова складність розв'язку TSP методом Bitmask DP для N міст?

a) $O(N)$

b) $O(N^2)$

c) $O(N^2 \cdot 2^N)$

d) $O(N \log N)$

5. У стані `dp[mask][i]`, що представляє `mask`?

a) Кількість відвіданих міст.

b) Підмножину відвіданих міст.

c) Поточну відстань.

d) Номер попереднього міста.

5. Базовий випадок для ДП-розв'язку TSP зазвичай ініціалізується як:

a) `dp[0][0] = 0`

b) `dp[1][0] = 0`

c) `dp[0][1] = 0`

d) Всі елементи таблиці ініціалізуються нулями.

Відповіді: 1-c, 2-b, 3-c, 4-b, 5-b

Завдання для самостійної роботи

1. Відновлення шляху в TSP. Модифікуйте програму з лабораторної роботи так, щоб вона виводила не лише довжину маршруту, а й саму послідовність міст.

2. Задача про призначення (Assignment Problem). Є N робітників та N завдань. Задано матрицю вартості `cost[i][j]`, де вказано, скільки коштує доручити *i*-му робітнику *j*-те завдання. Потрібно знайти розподіл завдань (один робітник — одне завдання) з мінімальною сумарною вартістю. Розв'яжіть за допомогою Bitmask DP.

3. Найкоротший гамільтонів шлях. Модифікуйте алгоритм для TSP, щоб він знаходив найкоротший шлях, що відвідує кожне місто рівно один раз, але не вимагає повернення у стартове місто.
4. Підрахунок гамільтонових шляхів. Напишіть програму, яка для заданого незваженого графа підраховує кількість різних гамільтонових шляхів, що починаються у вершині 0, за допомогою Bitmask DP.
5. Задача про покриття множини (Set Cover). Є універсальна множина U={1, ..., N} та набір підмножин $S_1, S_2, \ldots, S_k$. Потрібно знайти найменшу кількість підмножин S_i, об'єднання яких дає U. Розв'яжіть для малого N.
6. Задача про найдовший шлях у DAG. Хоча цю задачу можна розв'язати простіше, спробуйте розв'язати її за допомогою Bitmask DP. Порівняйте складність з класичним методом.
7. Задача про плитки (Tiling Problem). Скількома способами можна замостити прямокутник 2xN плитками доміно розміром 1x2? Розв'яжіть за допомогою Bitmask DP по профілю.
8. Оптимізація пам'яті. Чи можна оптимізувати використання пам'яті в алгоритмі для TSP? Поміркуйте, чи залежить обчислення стану для маски `m` від станів з масками, що мають меншу кількість встановлених бітів.
9. Bitmask DP на деревах. Дослідіть, як техніка Bitmask DP застосовується до задач на деревах (наприклад, для знаходження максимальної незалежної множини).
10. Зважена задача про максимальну незалежну множину. Напишіть розв'язок для цієї задачі на графі з N⩽20 вершинами, де кожна вершина має вагу.
11. Продавець меблів. У продавця є кілька предметів меблів, кожен з яких має довжину. Він хоче розмістити їх уздовж двох стін кімнати. Розробіть алгоритм, що мінімізує максимальну з довжин зайнятого простору вздовж цих двох стін.
12. Задача про «чисті» шляхи. У графі деякі вершини «брудні». Знайдіть найкоротший шлях з вершини А до Б, який проходить не більше ніж через `k` брудних вершин.

13. Розбиття рядка на паліндроми. Знайдіть мінімальну кількість паліндромних підрядків, на які можна розрізати заданий рядок.
14. Есе-порівняння. Напишіть есе, що порівнює розв'язки TSP методами повного перебору, гілок та меж і динамічного програмування. Оцініть їхню складність та практичну придатність.
15. Межі можливостей. Експериментально визначте, для якого максимального N ваша реалізація TSP на вашому комп'ютері працює прийнятний час (наприклад, до 1 хвилини).

Динамічне програмування по розрядах

Розглянемо техніку динамічного програмування, яка застосовується до цілого класу задач, пов'язаних із підрахунком чисел. Цей метод, відомий як ДП по розрядах (Digit DP), дозволяє ефективно розв'язувати задачі, що на перший погляд здаються надзвичайно складними.

1. Теоретичні відомості

1.1. Новий клас задач: підрахунок чисел із властивостями

Уявіть, що перед вами стоїть задача: «Скільки існує чисел в діапазоні від A до B, сума цифр яких є парною?». Перебирати кожне число і перевіряти його властивість — це можливо для малих діапазонів, але абсолютно неприйнятно, якщо A та B є числами з десятками знаків.

Тут нам на допомогу приходить техніка ДП по розрядах. Її центральна ідея базується на простому, але потужному спостереженні:

Кількість чисел з властивістю P в діапазоні `[A, B]` дорівнює `(кількість чисел з властивістю P в [0, B]) - (кількість чисел з властивістю P в [0, A-1])`.

Це дозволяє нам звести складну задачу про довільний діапазон до розв'язання однієї й тієї ж підзадачі: «Скільки чисел від 0 до R задовольняють умову P?». Ми навчимося писати функцію `count(R, P)`, а потім фінальний результат буде просто `count(B, P) - count(A-1, P)`.

1.2. Побудова числа по розрядах та простір станів

Отже, наша мета — написати функцію `count(R)`, яка рахує «хороші» числа від 0 до R. Замість того, щоб перебирати числа, ми будемо їх конструювати цифра за цифрою, зліва направо (від старших розрядів до молодших).

Цей процес побудови природно вкладається в рекурсивну структуру. Ми будемо писати рекурсивну функцію, яка намагається поставити цифру на

поточну позицію `index`, знаючи, які цифри були поставлені на попередніх позиціях.

Але що саме нам потрібно знати від попередніх кроків, щоб прийняти рішення зараз? Це приводить нас до визначення стану ДП. Для задач такого типу стан майже завжди включає:

1. `index` — поточна позиція (розряд), яку ми заповнюємо.
2. Параметри, що описують властивість, яку ми перевіряємо (наприклад, поточна сума цифр).
3. Один дуже важливий, «магічний» параметр, який ми розглянемо окремо.

1.3. Ключовий параметр: обмеження `is_less`

Уявіть, що нам потрібно порахувати кількість «хороших» чисел до R = 582. Ми починаємо будувати наше число зліва.

Позиція 0 (сотні): Яку цифру ми можемо поставити сюди? Будь-яку від 0 до 5.

– Варіант А: Ми ставимо цифру, що менша за відповідну цифру в R. Наприклад, ставимо `d=4`. Що це означає для решти цифр? Оскільки наш префікс «4» вже менший за префікс R «5», то на наступні позиції ми можемо ставити будь-які цифри від 0 до 9. Наше число гарантовано буде меншим за 582. Ми зняли верхнє обмеження.
– Варіант Б: Ми ставимо цифру, що дорівнює відповідній цифрі в R. Ставимо `d=5`. Наш префікс «5» дорівнює префіксу R. Це означає, що на наступному кроці ми знову будемо обмежені відповідною цифрою числа R (цифрою 8).

Саме для відстеження цієї ситуації ми вводимо булевий прапорець `is_less` (або `is_tight`):

– `is_less = false`: Означає, що префікс, який ми побудували, збігається з префіксом числа R. На поточній позиції ми можемо ставити цифри лише від 0 до `R[index]`.
– `is_less = true`: Означає, що наш префікс вже менший за префікс R. На поточній та всіх наступних позиціях ми можемо ставити будь-яку цифру від 0 до 9.

Цей прапорець передається від одного рекурсивного виклику до іншого, дозволяючи алгоритму знати, чи потрібно дотримуватися верхньої межі R.

1.4. Детальний розбір: Підрахунок чисел із заданою сумою цифр

Постановка задачі: Порахувати кількість чисел в діапазоні `[0, R]`, сума цифр яких дорівнює `S`.

Стан ДП: Нам потрібно знати, на якій ми позиції, чи є у нас обмеження зверху, і яка сума цифр у нас вже набрана. `dp[index][is_less][current_sum]` — кількість способів завершити число, починаючи з позиції `index`, маючи вже набрану суму `current_sum` та обмеження `is_less`.

Рекурентне співвідношення: `solve(index, is_less, current_sum)` — рекурсивна функція, що обчислює наш стан.

1. Базовий випадок: Якщо `index` дійшов до кінця числа, ми перевіряємо, чи `current_sum` дорівнює потрібній нам сумі `S`. Якщо так, ми знайшли одне «хороше» число, повертаємо 1. Інакше — 0.
2. Мемоізація: Перед обчисленням перевіряємо, чи не зберігається результат для стану `(index, is_less, current_sum)` у нашій ДП-таблиці. Якщо так — повертаємо його.
3. Рекурсивний крок:
    - Визначаємо верхню межу для поточної цифри: `limit = is_less ? 9 : R[index]`.
    - Запускаємо цикл для поточної цифри `d` від 0 до `limit`.
    - Усередині циклу робимо рекурсивний виклик для наступної позиції: `solve(index + 1, new_is_less, current_sum + d)`.
    - `new_is_less` дорівнюватиме `true`, якщо `is_less` вже був `true`, АБО якщо `d < limit`.
    - Сумуємо результати всіх рекурсивних викликів.
4. Зберігаємо обчислений результат у ДП-таблицю і повертаємо його.

Псевдокод:

```
// Глобальні змінні: R_str (число R у вигляді рядка), S (цільова сума),
// dp[][][] (таблиця для мемоізації, заповнена -1)

ФУНКЦІЯ solve(index, is_less, current_sum):
    // Базовий випадок: дійшли до кінця числа
    ЯКЩО index == довжина(R_str) ТО
```

```
        ПОВЕРНУТИ (ЯКЩО current_sum == S ТО 1 ІНАКШЕ 0)
    КІНЕЦЬ ЯКЩО

    // Мемоізація
    ЯКЩО dp[index][is_less][current_sum] != -1 ТО
        ПОВЕРНУТИ dp[index][is_less][current_sum]
    КІНЕЦЬ ЯКЩО

    // Рекурсивний крок
    count = 0
    limit = (ЯКЩО is_less ТО 9 ІНАКШЕ цифра(R_str[index]))

    ДЛЯ digit ВІД 0 ДО limit:
        // Перевіряємо, чи не перевищуємо ми цільову суму
        ЯКЩО current_sum + digit <= S ТО
            // Визначаємо новий прапорець is_less
            new_is_less = is_less || (digit < limit)
            // Робимо рекурсивний виклик
            count = count + solve(index + 1, new_is_less, current_sum
+ digit)
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    // Зберігаємо та повертаємо результат
    dp[index][is_less][current_sum] = count
    ПОВЕРНУТИ count
КІНЕЦЬ ФУНКЦІЇ

// Основна функція для запуску
ФУНКЦІЯ count_good_numbers(R_as_string):
    // ... ініціалізація dp таблиці ...
    ПОВЕРНУТИ solve(0, false, 0)
КІНЕЦЬ ФУНКЦІЇ
```

2. Область застосування

ДП по розрядах — це вузькоспеціалізована техніка, яка застосовується до комбінаторних задач підрахунку цілих чисел у заданому діапазоні, що задовольняють певним властивостям, які залежать від їхніх цифр.

- Спортивне програмування: Це основна сфера, де ця техніка є дуже популярною.
- Теоретична інформатика: Для аналізу комбінаторних властивостей чисел.

– Криптографія: В деяких задачах, пов'язаних з аналізом числових послідовностей.

3. Приклади застосування

– Підрахунок «щасливих» чисел: Знайти кількість чисел у діапазоні [A, B], що містять лише цифри 4 та 7.
– Числа без повторень: Знайти кількість чисел, у яких усі цифри є унікальними.
– Числа, що діляться на K: Знайти кількість чисел, що діляться на задане число K (тут до стану ДП додається залишок від ділення на K).

Переваги та недоліки

Переваги:

– Висока ефективність: Дозволяє розв'язувати задачі, які здаються експоненційними, за поліноміальний час. Складність зазвичай дорівнює **`O(довжина_числа * простір_станів)`**. Для нашого прикладу це приблизно $O(\log_{10}(R) \cdot 2 \cdot S)$.
– Універсальний шаблон: Загальна структура рекурсивної функції з мемоізацією є однаковою для більшості задач, змінюються лише додаткові параметри стану.

Недоліки:

– Вузька спеціалізація: Застосовується лише до дуже специфічного класу задач.
– Складність формулювання стану: Найскладніша частина — правильно визначити всі параметри, що мають входити до стану ДП.
– Обмеження на параметри: Якщо додаткові параметри стану можуть приймати багато значень (наприклад, сума цифр може бути великою), то ДП-таблиця стає занадто великою.

Контрольні запитання

1. Поясніть своїми словами, чому задача підрахунку в діапазоні [A, B] зводиться до **`count(B) - count(A-1)`**.
2. Яку роль відіграє прапорець **`is_less`** в алгоритмі ДП по розрядах?
3. Опишіть, як би ви сформулювали стан ДП для задачі «порахувати кількість чисел до R, що не містять двох однакових цифр поспіль».

4. Чому базовим випадком рекурсії є досягнення кінця числа (`index == довжина(R)`)?
5. Яка головна перевага ДП по розрядах над простим перебором чисел від A до B?

Методичні вказівки до лабораторної роботи

1. Назва

Підрахунок чисел із заданою сумою цифр методом динамічного програмування по розрядах.

2. Мета

Практичне засвоєння техніки ДП по розрядах (Digit DP) шляхом реалізації рекурсивного розв'язку з мемоізацією для задачі комбінаторного підрахунку.

3. Завдання

Створити програму, яка для заданого діапазону [A, B] та цільової суми S знаходить кількість чисел у цьому діапазоні, сума цифр яких дорівнює S.

4. Опис

Програма має реалізовувати функцію `count(R, S)`, яка підраховує кількість «хороших» чисел в діапазоні `[0, R]`. Фінальний результат для `[A, B]` обчислюється як `count(B, S) - count(A-1, S)`. Основна логіка має бути реалізована у вигляді рекурсивної функції з мемоізацією.

5. Порядок виконання

1. Створіть головну функцію: Напишіть функцію, яка приймає на вхід A, B, S і повертає `count(B_str, S) - count(A_minus_1_str, S)`.
2. Реалізуйте рекурсивну функцію `solve`: Керуючись псевдокодом, напишіть рекурсивну функцію, яка приймає поточний індекс, прапорець `is_less` та поточну суму.
3. Налаштуйте мемоізацію: Створіть 3-вимірну ДП-таблицю та ініціалізуйте її значеннями, що позначають необчисленість (напр., -1). Додайте перевірку наявності результату на початку функції та збереження результату в кінці.
4. Реалізуйте логіку циклу: Всередині рекурсивної функції напишіть цикл по цифрах від 0 до `limit`, де `limit` залежить від `is_less`.

5. Обробіть рекурсивні виклики: Правильно обчислюйте `new_is_less` та `new_current_sum` для рекурсивних викликів.
6. Протестуйте: Перевірте роботу на невеликих діапазонах, для яких можна порахувати відповідь вручну (напр., кількість чисел до 100 з сумою цифр 5).

6. Результат

Додаток, що коректно обчислює кількість чисел із заданою сумою цифр у великих діапазонах.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з реалізацією рекурсивної функції з мемоізацією.
2. Продемонструвати роботу додатку на кількох тестових діапазонах [A, B] та сумах S.
3. Бути готовим відповісти на питання:
    - Пояснити параметри стану вашої ДП.
    - Як працює прапорець `is_less` у вашому коді?
    - Яка максимальна довжина числа та максимальна сума цифр, які може обробити ваша програма, виходячи з розміру ДП-таблиці?

Тести

1. Задача підрахунку чисел в діапазоні [A, B] зазвичай зводиться до:

a) `count(B) - count(A)`

b) `count(B) + count(A)`

c) `count(B) - count(A-1)`

d) `count(B-A)`

2. У ДП по розрядах прапорець `is_less = true` означає, що:

a) Поточна цифра менша за 9.

b) Побудований нами префікс вже менший за префікс верхньої межі, тому далі можна ставити будь-які цифри.

c) Побудований нами префікс дорівнює префіксу верхньої межі.

d) Сума цифр менша за цільову.

3. Який підхід до ДП є найбільш природним для реалізації ДП по розрядах?

a) Табуляція (Bottom-up).

b) Рекурсія з мемоізацією (Top-down).

c) Жадібний підхід.

d) Пошук з поверненням.

4. Якою буде складність розв'язку задачі «порахувати кількість чисел до R (довжиною D), що не містять цифру 5»?

a) $O(D)$

b) $O(D^2)$

c) $O(10^D)$

d) $O(D \cdot 10)$

5. Стан ДП для задачі «порахувати кількість чисел до R, що діляться на K» найімовірніше буде:

a) `dp[index][is_less]`

b) `dp[index][is_less][current_sum]`

c) `dp[index][is_less][current_remainder_mod_K]`

d) `dp[index][current_sum][current_remainder_mod_K]`

Відповіді: 1-c, 2-b, 3-b, 4-a (оскільки простір станів для «наявність 5» дуже малий), 5-c

Завдання для самостійної роботи

1. Числа без сусідніх дублікатів. Напишіть програму, що рахує кількість чисел в діапазоні [A, B], в яких жодні дві сусідні цифри не є однаковими.
2. «Гірські» числа. «Гірським» назвемо число, цифри якого спочатку строго зростають, а потім строго спадають (напр., 13864). Послідовність може тільки зростати або тільки спадати. Порахуйте кількість таких чисел до N.
3. Паліндроми. Порахуйте кількість чисел-паліндромів у заданому діапазоні [A, B].
4. Збалансовані числа. Назвемо число збалансованим, якщо сума цифр на парних позиціях дорівнює сумі цифр на непарних. Порахуйте кількість таких чисел.
5. Кількість нулів. Порахуйте загальну кількість нулів, що використовуються для запису всіх чисел від A до B.

6. Кратність K. Порахуйте кількість чисел в діапазоні [A, B], що діляться на K.
7. Сума сум. Знайдіть суму сум цифр для всіх чисел в діапазоні [A, B].
8. Числа з рівною кількістю 3, 6, 9. Порахуйте кількість чисел, у яких кількість цифр ‘3’ дорівнює кількості ‘6’ і дорівнює кількості ‘9’.
9. Двійкові рядки без «11». Порахуйте кількість двійкових рядків довжиною N, що не містять підрядка «11». Розв’яжіть класичним ДП, а потім спробуйте адаптувати підхід Digit DP.
10. Зростаючі числа. Порахуйте кількість чисел, цифри яких утворюють строго зростаючу послідовність (напр., 1358).
11. Числа з дільником X. Порахуйте кількість чисел до N, які діляться на свій власний набір цифр (наприклад, 12 ділиться на 1 та 2).
12. Есе-аналіз. Напишіть есе, що порівнює Bitmask DP та Digit DP. Що в них спільного (використання ДП для розв’язання комбінаторних задач на невеликих вхідних даних)? У чому їхня принципова відмінність?
13. Задача з двома обмеженнями. Порахуйте кількість чисел у діапазоні [A, B], сума цифр яких лежить в діапазоні [S1, S2].
14. Прості суми. Порахуйте кількість чисел до N, сума цифр яких є простим числом.
15. Задача на іншу систему числення. Адаптуйте будь-яке з попередніх завдань (наприклад, про суму цифр) для шістнадцяткової системи числення.

Пошук з поверненням.

Ми розглянули алгоритми, які роблять один «впевнений» вибір (жадібні) та ті, що розв’язують всі підзадачі для побудови оптимального рішення (ДП). Тепер ми познайомимось із парадигмою, яка є систематичним методом проб і помилок, що дозволяє знаходити розв’язки у величезному просторі варіантів.

1. Теоретичні відомості

1.1. Вступ: Пошук у просторі станів

Уявіть, що ви стоїте на вході у великий і заплутаний лабіринт. Ваша мета — знайти вихід. Якою буде ваша стратегія? Ви, ймовірно, підете одним із шляхів. На кожному роздоріжжі ви будете обирати один з можливих

напрямків. Якщо ви зайшли у глухий кут, ви не будете стояти на місці. Ви повернетесь до останнього роздоріжжя, де у вас був вибір, і спробуєте інший шлях.

Цей процес — рух уперед, доки це можливо, і повернення назад у разі невдачі — є ідеальною аналогією для пошуку з поверненням.

Багато складних задач, як-от розв'язання головоломок, знаходження комбінацій або виграшних стратегій, можна уявити як пошук правильного шляху у величезному дереві простору станів. Кожен вузол цього дерева — це певний проміжний стан (наприклад, частково заповнена шахова дошка), а гілки — це можливі кроки або рішення. Перебрати абсолютно всі шляхи (метод «грубої сили») зазвичай неможливо через їхню астрономічну кількість.

Пошук з поверненням — це розумний спосіб обрізати це дерево пошуку. Ми будуємо розв'язок крок за кроком. Якщо на поточному кроці ми бачимо, що обраний шлях гарантовано не приведе до успіху, ми «обрізаємо» всю цю гілку дерева і повертаємось назад, щоб спробувати інший варіант.

1.2. Загальна структура алгоритму з поверненням

Більшість алгоритмів пошуку з поверненням можна описати за допомогою єдиного рекурсивного шаблону. Це робить їх дуже потужним інструментом.

Ідея: Ми маємо функцію, яка намагається розширити поточний частковий розв'язок.

1. Функція отримує поточний стан (наприклад, частково побудовану конфігурацію).
2. Базовий випадок: Якщо поточний стан є повним і задовольняє всім умовам задачі, ми знайшли розв'язок! Ми обробляємо його (наприклад, виводимо на екран або додаємо до списку знайдених) і повертаємось.
3. Рекурсивний крок: Ми перебираємо всі можливі «ходи» або «вибори», які можна зробити з поточного стану.
    - Для кожного можливого ходу:
        d. Зробити хід: Додаємо вибір до нашого часткового розв'язку.

e. Перевірка: Якщо новий стан все ще є перспективним (не порушує жодних правил), ми рекурсивно викликаємо нашу функцію для цього нового, розширеного стану.

f. Повернутися назад (Backtrack): Це ключовий крок! Ми скасовуємо зроблений хід, видаляючи його з нашого часткового розв'язку. Це дозволяє нам повернутися до попереднього стану і спробувати наступний можливий хід з тієї ж точки.

1.3. Детальний розбір: Задача N-ферзів

Це одна з найвідоміших задач, що ідеально ілюструє пошук з поверненням. Постановка задачі: Розмістити N шахових ферзів на дошці розміром $N \times N$ так, щоб жодні два ферзі не загрожували один одному (тобто не стояли на одній горизонталі, вертикалі чи діагоналі).

Формулювання для пошуку з поверненням: Ми будемо розміщувати ферзів по одному в кожен стовпець, починаючи з першого.

- Стан: Положення ферзів у вже пройдених стовпцях.
- Вибір: У яку клітинку (рядок) поточного стовпця поставити наступного ферзя?
- Обмеження: Новий ферзь не повинен потрапляти під удар вже розміщених ферзів.

Псевдокод розв'язання задачі N-ферзів:

```
// Основна функція, що запускає процес
ФУНКЦІЯ solveNQueens(N):
    // Створюємо порожню дошку N x N
    board = нова_матриця(N, N, заповнена 0)

    // Викликаємо рекурсивного помічника для першого стовпця
    ЯКЩО solveNQueensUtil(board, 0) == false ТО
        ВИВЕСТИ "Розв'язку не існує"
    КІНЕЦЬ ЯКЩО
КІНЕЦЬ ФУНКЦІЇ

// Рекурсивна функція-помічник
ФУНКЦІЯ solveNQueensUtil(board, col):
    // Базовий випадок: якщо ми розмістили ферзів у всіх стовпцях,
    // ми знайшли розв'язок.
    N = розмірність(board)
    ЯКЩО col >= N ТО
```

```
        ВИВЕСТИ_РОЗВ'ЯЗОК(board)
        ПОВЕРНУТИ true // Можна зупинитись, якщо потрібен один розв'я
зок
    КІНЕЦЬ ЯКЩО

    // Рекурсивний крок: пробуємо поставити ферзя в кожен рядок поточ
ного стовпця 'col'
    ДЛЯ row ВІД 0 ДО N-1:
        // Перевіряємо, чи безпечно ставити ферзя сюди
        ЯКЩО isSafe(board, row, col) ТО
            // 1. Робимо хід: ставимо ферзя
            board[row][col] = 1

            // Рекурсивно викликаємо для наступного стовпця
            solveNQueensUtil(board, col + 1)

            // 2. Повертаємось назад (Backtrack): прибираємо ферзя
            // Це дозволить спробувати поставити його в інший рядок н
а наступній ітерації циклу
            board[row][col] = 0
        КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ
КІНЕЦЬ ФУНКЦІЇ

// Функція перевірки безпечності позиції
ФУНКЦІЯ isSafe(board, row, col):
    // Перевіряємо горизонталь ліворуч
    ДЛЯ i ВІД 0 ДО col-1:
        ЯКЩО board[row][i] == 1 ТО ПОВЕРНУТИ false КІНЕЦЬ ЯКЩО
    КІНЕЦЬ ДЛЯ

    // Перевіряємо верхню ліву діагональ
    ДЛЯ i=row, j=col ПОКИ i >= 0 І j >= 0:
        ЯКЩО board[i][j] == 1 ТО ПОВЕРНУТИ false КІНЕЦЬ ЯКЩО
        i = i - 1; j = j - 1
    КІНЕЦЬ ПОКИ

    // Перевіряємо нижню ліву діагональ
    ДЛЯ i=row, j=col ПОКИ i < N І j >= 0:
        ЯКЩО board[i][j] == 1 ТО ПОВЕРНУТИ false КІНЕЦЬ ЯКЩО
        i = i + 1; j = j - 1
    КІНЕЦЬ ПОКИ

    ПОВЕРНУТИ true
КІНЕЦЬ ФУНКЦІЇ
```

1.4. Порівняння з іншими парадигмами

- Пошук з поверненням vs. »Груба сила»: Пошук з поверненням — це значно оптимізована версія «грубої сили». Замість того, щоб генерувати всі можливі розстановки ферзів і потім перевіряти їх, ми відкидаємо неправильні варіанти на ранніх етапах.
- Пошук з поверненням vs. Жадібні алгоритми: Жадібний алгоритм робить один локально найкращий вибір і ніколи не повертається назад. Пошук з поверненням, навпаки, є методом дослідження всіх перспективних шляхів, систематично повертаючись назад, щоб спробувати альтернативи.
- Пошук з поверненням vs. Динамічне програмування: ДП зазвичай застосовується до оптимізаційних задач з властивістю перекриття підзадач (де ми зберігаємо результати, щоб не обчислювати їх знову). Пошук з поверненням частіше використовується для задач задоволення обмежень, задач пошуку або переліку, де кожен шлях у дереві пошуку є унікальним і запам’ятовувати розв’язки підзадач немає сенсу.

2. Область застосування

Пошук з поверненням є універсальним методом, що застосовується до широкого кола комбінаторних задач.

- Головоломки та ігри: Розв’язання Судоку, пошук виходу з лабіринту, розв’язання кросвордів.
- Задачі задоволення обмежень (Constraint Satisfaction Problems): Задача N-ферзів, задача розфарбовування графа.
- Комбінаторний пошук та генерація: Генерація всіх перестановок, комбінацій або підмножин заданої множини.
- Синтаксичний аналіз (Parsing): Може використовуватися для розбору виразів у мовах програмування.

3. Приклади застосування

- Генератори паролів: Програми, що генерують паролі за певними критеріями (наявність великих літер, цифр тощо), можуть використовувати пошук з поверненням для побудови всіх можливих варіантів.

– Штучний інтелект в іграх: Пошук виграшної послідовності ходів у таких іграх, як шахи або хрестики-нулики (для невеликих дощок), є задачею пошуку в дереві станів.
– Регулярні вирази: Деякі рушії регулярних виразів використовують пошук з поверненням для знаходження відповідностей складним патернам.

Переваги та недоліки

Переваги:

– Універсальність: Шаблон алгоритму можна адаптувати до величезної кількості різноманітних задач.
– Гарантоване знаходження розв'язку: На відміну від евристик, пошук з поверненням є повним методом — якщо розв'язок існує, він його знайде.
– Значно ефективніший за «грубу силу» завдяки механізму «обрізання» дерева пошуку.

Недоліки:

– Експоненційна складність: У найгіршому випадку, коли «обрізати» дерево не вдається, алгоритм може перебрати майже всі можливі варіанти, що призводить до експоненційної або факторіальної складності.
– Не для оптимізації: Класичний пошук з поверненням призначений для знаходження *будь-якого* або *всіх* розв'язків, а не *найкращого*. Для оптимізації його розширюють до методу гілок та меж (Branch and Bound).

Контрольні запитання

4. Опишіть своїми словами, що таке «простір станів» задачі.
5. У чому полягає ключова відмінність між пошуком з поверненням та методом «грубої сили»?
6. Поясніть роль кроку «Повернутися назад (Backtrack)» у загальному шаблоні алгоритму. Що станеться, якщо його пропустити?
7. Чому динамічне програмування зазвичай не застосовують до задачі N-ферзів?

8. Наведіть приклад задачі, де жадібний алгоритм дасть швидку, але, можливо, неправильну відповідь, тоді як пошук з поверненням знайде правильну, хоча й повільніше.

Методичні вказівки до лабораторної роботи

1. Назва

Розв'язання задачі N-ферзів за допомогою алгоритму пошуку з поверненням.

2. Мета

Практичне засвоєння парадигми пошуку з поверненням шляхом реалізації рекурсивного алгоритму для класичної задачі задоволення обмежень.

3. Завдання

Створити програму, яка знаходить та виводить на екран одне з можливих рішень для задачі N-ферзів для заданого *N*.

4. Опис

Програма має представляти шахову дошку у вигляді двовимірного масиву. Основна логіка має бути реалізована у вигляді рекурсивної функції, яка намагається розмістити ферзів по стовпцях.

5. Порядок виконання

1. Створіть представлення дошки: Ініціалізуйте двовимірний масив $N \times N$, заповнений нулями.
2. Реалізуйте функцію перевірки: Напишіть функцію `isSafe(board, row, col)`, яка перевіряє, чи можна ставити ферзя у вказану позицію, аналізуючи горизонталі та діагоналі ліворуч від поточного стовпця.
3. Реалізуйте рекурсивну функцію: Керуючись псевдокодом, напишіть основну функцію `solveNQueensUtil(board, col)`.
4. Організуйте виклик: Напишіть головну функцію `solveNQueens(N)`, яка створює дошку та запускає рекурсивний процес з першого стовпця (`col=0`).
5. Виведіть результат: Модифікуйте базовий випадок рекурсії так, щоб при знаходженні розв'язку він виводив матрицю дошки на екран.

6. Результат

Додаток, що для заданого *N* (наприклад, N=8) виводить на екран коректну розстановку ферзів.

7. Після виконання лабораторної роботи студент повинен надати:
    1. Вихідний код програми, де чітко реалізована рекурсивна функція та функція перевірки безпечності.
    2. Продемонструвати роботу додатку для різних значень N (напр., 4, 5, 8).
    3. Бути готовим відповісти на питання:
        - Пояснити, як працює ваш рекурсивний алгоритм і де саме відбувається «повернення назад».
        - Чому у функції `isSafe` ми перевіряємо лише ліву частину дошки?
        - Як можна модифікувати ваш код, щоб він знаходив не один, а всі можливі розв’язки?

Тести

1. Пошук з поверненням — це, по суті, оптимізований…

uuuuu) Жадібний алгоритм.

vvvvv) Метод динамічного програмування.

wwwww) Повний перебір (brute-force).

xxxxx) Алгоритм Дейкстри.

2. Який крок є ключовим у реалізації пошуку з поверненням і відрізняє його від простого рекурсивного перебору?

yyyyy) Перевірка базового випадку.

zzzzz) Цикл по можливих ходах.

aaaaaa) Скасування зробленого ходу після повернення з рекурсії.

bbbbbb) Виклик функції самої себе.

3. Для якого типу задач пошук з поверненням є найбільш природним вибором?

cccccc) Обчислення математичних функцій.

dddddd) Сортування масивів.

eeeeee) Задачі задоволення обмежень та комбінаторного пошуку.

ffffff) Пошук найкоротшого шляху в графі.

4. У найгіршому випадку часова складність алгоритму пошуку з поверненням може бути:

gggggg) Лінійною.

hhhhhh) Логарифмічною.

iiiiii) Поліноміальною.

jjjjjj) Експоненційною.

5. Розв’язуючи задачу N-ферзів, на етапі розміщення ферзя у стовпець `k`, функція `isSafe` перевіряє конфлікти з ферзями у стовпцях:

kkkkkk) Від `k+1` до `N-1`.

llllll) У всіх інших стовпцях.

mmmmmm) Лише у стовпці `k-1`.

nnnnnn) Від `0` до `k-1`.

Відповіді: 1-c, 2-c, 3-c, 4-d, 5-d

Завдання для самостійної роботи

1. Розв’язувач Судоку. Напишіть програму, яка розв’язує будь-яку головоломку Судоку 9x9 за допомогою пошуку з поверненням.
2. Генератор перестановок. Напишіть функцію, яка генерує та виводить усі можливі перестановки елементів заданого масиву.
3. Генератор підмножин. Напишіть функцію, яка генерує та виводить усі можливі підмножини (powerset) заданої множини.
4. Вихід з лабіринту. Представте лабіринт у вигляді двовимірної матриці. Напишіть програму, що знаходить шлях від стартової до кінцевої точки.
5. Задача про розфарбовування графа. Для заданого графа та k кольорів визначте, чи можна розфарбувати його вершини так, щоб жодні дві суміжні вершини не мали однакового кольору.
6. Задача комівояжера. Використовуйте пошук з поверненням (в рамках методу гілок та меж), щоб знайти найкоротший гамільтонів цикл у невеликому зваженому графі.
7. Задача про суму підмножини (Subset Sum). Для заданої множини цілих чисел та цільової суми S знайдіть підмножину, сума елементів якої дорівнює S.
8. Комбінації. Напишіть функцію, що генерує всі можливі комбінації з k елементів із заданої множини з n елементів.

9. Правильні дужкові послідовності. Напишіть функцію, що генерує всі правильні дужкові послідовності довжиною 2*n.
10. Задача про хід коня. Напишіть програму, що знаходить послідовність ходів шахового коня, яка дозволяє йому відвідати кожну клітинку дошки $N \times N$ рівно один раз.

Метод гілок та меж

Ми підійшли до потужного розширення ідеї пошуку з поверненням. Якщо пошук з поверненням шукає *будь-який* розв'язок, то метод гілок та меж шукає *найкращий*. Цей розділ покаже, як додати «інтелект» до нашого перебору, щоб відсікати не просто неможливі, а й завідомо неоптимальні шляхи.

1. Теоретичні відомості

1.1. Від пошуку до оптимізації: еволюція пошуку з поверненням

У попередньому розділі ми розглянули пошук з поверненням — потужний метод для розв'язання задач, де потрібно знайти розв'язок, що задовольняє певним обмеженням, наприклад, розставити ферзів на шаховій дошці. Алгоритм систематично досліджував дерево можливих станів, відкидаючи гілки, що вели в глухий кут.

Але що, якби кожна розстановка ферзів мала певну «вартість» і нам потрібно було б знайти розстановку з *мінімальною* вартістю? Або, уявіть лабіринт, де кожен коридор має свою довжину, і нам потрібен не просто *будь-який* вихід, а *найкоротший* шлях до нього? Це вже задачі оптимізації.

Простий пошук з поверненням тут безсилий, бо він змушений буде перебрати абсолютно всі можливі розв'язки, щоб потім порівняти їх і обрати найкращий. Метод гілок та меж (Branch and Bound) є еволюцією пошуку з поверненням, створеною спеціально для таких оптимізаційних задач. Його ключова ідея — «обрізати» не тільки безперспективні, але й завідомо неоптимальні гілки дерева пошуку.

1.2. Три кити методу «Гілок та меж» (B&B)

B&B, як і пошук з поверненням, будує дерево станів, але додає до цього процесу два нові потужні інструменти: оцінку меж та стратегію вибору.

11. Розгалуження. Це той самий процес, що й у пошуку з поверненням. Ми беремо поточну задачу (вузол дерева) і розбиваємо

її на декілька менших підзадач (генеруємо дочірні вузли). Наприклад, для задачі комівояжера, якщо ми знаходимось у місті A, ми створюємо гілки для переходу в кожне з іще не відвіданих міст.

12. Оцінка меж. Це серце методу. Для кожної згенерованої підзадачі (вузла) ми обчислюємо межу — «оптимістичну» оцінку найкращого розв'язку, який можна отримати, рухаючись по цій гілці.
    - Для задачі мінімізації (напр., пошук найкоротшого шляху), межа — це нижня оцінка. Вона каже нам: «найкращий розв'язок у цій гілці буде *не меншим* за це значення».
    - Для задачі максимізації (напр., задача про рюкзак), межа — це верхня оцінка (upper bound).

    Ця оцінка дозволяє нам виконувати відсікання. Ми постійно зберігаємо найкращий розв'язок, знайдений на даний момент (назвемо його «рекордом»). Якщо на якомусь етапі ми бачимо, що оптимістична оцінка для нової гілки гірша за наш поточний рекорд (наприклад, нижня межа шляху вже більша за довжину знайденого маршруту), то немає жодного сенсу досліджувати цю гілку далі. Ми її «обрізаємо».

13. Стратегія вибору. На відміну від простого DFS-обходу в пошуку з поверненням, у B&B ми маємо множину «активних» вузлів (підзадач), які чекають на дослідження. Стратегія вибору визначає, яку підзадачу ми візьмемо наступною.
    - LIFO (Last-In, First-Out): Використовується стек. Пошук схожий на DFS. Просто реалізувати, але можна заглибитись у неперспективну гілку.
    - FIFO (First-In, First-Out): Використовується черга. Пошук схожий на BFS. Досліджує дерево по рівнях.
    - Best-First Search (Пошук за найкращою оцінкою): Використовується черга з пріоритетами. На кожному кроці ми обираємо для дослідження вузол з найкращою (найбільш оптимістичною) межею. Ця стратегія зазвичай найшвидше призводить до оптимального розв'язку.

1.3. Детальний розбір: Задача комівояжера (TSP)

Це класична NP-складна задача оптимізації. Постановка задачі: Є множина міст та відстані між кожною парою. Потрібно знайти найкоротший

можливий маршрут, який відвідує кожне місто рівно один раз і повертається у вихідне місто.

Формулювання для B&B:

- Дерево простору станів: Корінь — це початкова точка без пройденого шляху. Кожен вузол представляє частково побудований маршрут. Наприклад, вузол `(A, C, D)` означає, що комівояжер пройшов з A в C, а потім в D.
- Розгалуження: З вузла `(A, C, D)` ми можемо створити дочірні вузли для кожного ще не відвіданого міста.
- Оцінка меж (Bounding Function): Це найскладніше і найважливіше. Як оцінити мінімально можливу довжину повного маршруту, маючи лише його частину?
    - Поточна вартість: довжина вже пройденого шляху.
    - Оптимістична оцінка залишку: нам потрібна нижня межа довжини шляху, щоб з'єднати всі невідвідані вершини і повернутись до стартової. Проста, але ефективна оцінка: для кожної невідвіданої вершини знайти суму довжин двох найдешевших ребер, що виходять з неї. Загальну суму цих довжин поділити на 2. Чому це працює? Бо в будь-якому повному циклі кожна вершина з'єднана рівно двома ребрами. Беручи найдешевші, ми отримуємо гарантовано не більшу оцінку, ніж буде у будь-якому реальному маршруті.
    - `bound = (поточна вартість шляху) + (оптимістична оцінка залишку)`

1.4. Псевдокод алгоритму для TSP

Використаємо стратегію Best-First Search з чергою з пріоритетами.

```
ФУНКЦІЯ solve_tsp_bnb(distance_matrix):
    // Ініціалізуємо чергу з пріоритетами для зберігання активних вуз
лів.
    // Пріоритет вузла - це його нижня межа (чим менше, тим краще).
    pq = нова_черга_з_пріоритетами()

    // Створюємо початковий вузол
    root_path = [0] // Починаємо з міста 0
    root_cost = 0
    root_bound = calculate_bound(root_path, distance_matrix)
```

```
    root_node = створити_вузол(path=root_path, cost=root_cost, bound=
root_bound)
    Додати(pq, root_node)

    // Ініціалізуємо "рекорд" дуже великим числом
    best_solution_cost = +∞
    best_solution_path = []

    // Основний цикл
    ПОКИ pq не порожня:
        // Вибираємо вузол з найкращою (найменшою) межею
        current_node = Видалити_мін(pq)

        // Відсікання: якщо навіть оптимістична оцінка гірша за знайд
ений розв'язок,
        // ігноруємо цю гілку.
        ЯКЩО current_node.bound >= best_solution_cost ТО
            ПРОДОВЖИТИ_ЦИКЛ
        КІНЕЦЬ ЯКЩО

        // Розгалуження: генеруємо дочірні вузли
        last_city = current_node.path.останній_елемент()
        ДЛЯ next_city ВІД 0 ДО N-1:
            ЯКЩО next_city не в current_node.path:
                // Створюємо новий частковий шлях
                new_path = current_node.path + [next_city]
                new_cost = current_node.cost + distance_matrix[last_c
ity][next_city]

                // Перевіряємо, чи це повний розв'язок
                ЯКЩО довжина(new_path) == N:
                    // Завершуємо цикл, додаючи шлях до початкового м
іста
                    total_cost = new_cost + distance_matrix[next_city
][0]
                    // Якщо цей розв'язок кращий за наш рекорд, оновл
юємо його
                    ЯКЩО total_cost < best_solution_cost:
                        best_solution_cost = total_cost
                        best_solution_path = new_path + [0]
                    КІНЕЦЬ ЯКЩО
                ІНАКШЕ:
                    // Це ще не повний шлях, обчислюємо межу і додаєм
о в чергу
```

```
                    new_bound = calculate_bound(new_path, new_cost, d
istance_matrix)
                    // Ще одне відсікання: додаємо вузол, тільки якщо
він перспективний
                    ЯКЩО new_bound < best_solution_cost:
                        new_node = створити_вузол(path=new_path, cost
=new_cost, bound=new_bound)
                        Додати(pq, new_node)
                    КІНЕЦЬ ЯКЩО
                КІНЕЦЬ ЯКЩО
        КІНЕЦЬ ДЛЯ
    КІНЕЦЬ ПОКИ

    ПОВЕРНУТИ best_solution_path, best_solution_cost
КІНЕЦЬ ФУНКЦІЇ
```

2. Область застосування

Метод гілок та меж є одним з найбільш відомих методів для розв'язання NP-складних задач оптимізації.

- Цілочисельне програмування: Пошук оптимальних розв'язків, де змінні можуть приймати лише цілі значення.
- Логістика та планування: Задача комівояжера, задача маршрутизації транспорту.
- Розподіл ресурсів: Оптимальний розподіл завдань між виконавцями.
- Проєктування мікросхем.

3. Приклади застосування

- Авіакомпанії: Оптимізація розкладу польотів та екіпажів для мінімізації витрат.
- Служби доставки: Побудова оптимальних маршрутів для кур'єрів, щоб мінімізувати загальний пробіг.
- Промислове виробництво: Планування послідовності операцій на верстатах для мінімізації часу простою.

Переваги та недоліки

Переваги:

- Знаходить доведений оптимум: На відміну від евристик, B&B гарантовано знаходить найкращий розв'язок.

– Значно ефективніший за повний перебір: Механізм оцінки меж дозволяє відсікати величезні частини дерева пошуку, роблячи розв’язання великих екземплярів задачі можливим.
– Гнучкість: Метод можна адаптувати до різних задач, розробляючи специфічні функції розгалуження та оцінки меж.

Недоліки:

– Експоненційна складність у найгіршому випадку: Якщо функція оцінки меж є слабкою і не відсікає гілки ефективно, B&B може виродитись до повного перебору.
– Залежність від якості оцінки: Ефективність алгоритму критично залежить від того, наскільки «тугою» (близькою до реального значення) є нижня (або верхня) межа. Розробка хорошої функції оцінки — це часто ціле мистецтво.
– Вимоги до пам’яті: Стратегії пошуку, як-от BFS або Best-First, можуть вимагати значного обсягу пам’яті для зберігання списку активних вузлів.

Контрольні запитання

1. У чому полягає ключова відмінність методу гілок та меж від пошуку з поверненням?
2. Поясніть роль «рекорду» (найкращого знайденого розв’язку) в процесі відсікання гілок.
3. Що таке «нижня межа» (lower bound) для задачі мінімізації і чому вона має бути «оптимістичною»?
4. Порівняйте стратегії пошуку LIFO та Best-First для методу гілок та меж. Яка з них, ймовірно, знайде оптимальний розв’язок швидше і чому?
5. Чому якість функції оцінки меж є настільки важливою для продуктивності алгоритму?

Методичні вказівки до лабораторної роботи

1. Назва

Розв’язання задачі комівояжера методом гілок та меж.

2. Мета

Практичне засвоєння методу гілок та меж шляхом реалізації алгоритму для класичної NP-складної задачі оптимізації. Розробка та застосування функції оцінки меж для відсікання простору пошуку.

3. Завдання

Створити програму, яка знаходить найкоротший гамільтонів цикл (розв'язок задачі комівояжера) для заданого зваженого графа, використовуючи метод гілок та меж зі стратегією пошуку Best-First.

4. Опис

Граф можна представити у вигляді матриці суміжності, де `matrix[i][j]` — відстань між містами `i` та `j`. Основна логіка буде зосереджена навколо черги з пріоритетами, яка зберігатиме активні вузли (часткові шляхи), відсортовані за їхньою нижньою межею.

5. Порядок виконання

1. Створіть представлення даних: Опишіть структуру або клас для вузла пошуку, що має містити: поточний шлях, поточну вартість шляху та обчислену нижню межу.
2. Реалізуйте функцію оцінки меж: Напишіть функцію `calculate_bound`, яка для заданого часткового шляху обчислює його нижню межу за описаним у теорії методом (або іншим на ваш вибір).
3. Ініціалізуйте процес: Створіть чергу з пріоритетами та додайте до неї початковий вузол (стартове місто).
4. Реалізуйте головний цикл: Напишіть цикл `while`, який дістає з черги вузол з найкращою межею, перевіряє умову відсікання та виконує розгалуження.
5. Обробляйте розв'язки: Коли довжина шляху дорівнює кількості міст, завершуйте цикл та оновлюйте «рекорд», якщо знайдений шлях є кращим.
6. Протестуйте: Перевірте роботу на невеликому графі (4-5 міст), для якого можна знайти оптимальний розв'язок вручну.

6. Результат

Додаток, що знаходить гарантовано оптимальний маршрут для задачі комівояжера та виводить сам маршрут і його довжину.

7. Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми з чіткою реалізацією структури вузла, функції оцінки меж та головного циклу B&B.
2. Продемонструвати роботу додатку на кількох тестових графах.
3. Бути готовим відповісти на питання:
    - Як працює ваша функція оцінки меж і чому вона є коректною нижньою оцінкою?
    - Як змінюється кількість досліджених вузлів, якщо зробити функцію оцінки меж гіршою (наприклад, завжди повертати 0)?
    - Яка структура даних лежить в основі вашої черги з пріоритетами?

Тести

1. Метод гілок та меж є розширенням…

ooooo) Жадібних алгоритмів для задач оптимізації.

ppppp) Пошуку з поверненням для задач оптимізації.

qqqqq) Динамічного програмування для задач пошуку.

rrrrr) Методу «Розділяй і володарюй» для задач задоволення обмежень.

2. Основна мета функції оцінки меж (bounding function) у задачі мінімізації — це:

ssssss) Точно обчислити вартість решти шляху.

ttttt) Надати песимістичну (верхню) оцінку вартості.

uuuuuu) Надати оптимістичну (нижню) оцінку вартості для відсікання неперспективних гілок.

vvvvvv) Визначити наступний найкращий хід.

3. Гілка дерева пошуку відсікається, якщо:

wwwww) Її нижня межа більша або дорівнює вартості найкращого вже знайденого розв’язку.

xxxxxx) Її нижня межа менша за вартість найкращого вже знайденого розв’язку.

yyyyyy) Вона містить більше 10 вузлів.

zzzzzz) Вона є занадто довгою.

4. Яка стратегія пошуку в методі B&B зазвичай найшвидше призводить до оптимального розв’язку?

aaaaaaa) Пошук в глибину (DFS).

bbbbbbb) Пошук в ширину (BFS).

ccccccc) Випадковий пошук.

ddddddd) Пошук за найкращою оцінкою (Best-First Search).

5. Метод гілок та меж гарантує знаходження:

eeeeeee) Будь-якого розв’язку, але дуже швидко.

fffffff) Наближеного до оптимального розв’язку.

ggggggg) Гарантовано оптимального розв’язку.

hhhhhhh) Лише одного розв’язку, навіть якщо їх декілька.

Відповіді: 1-b, 2-c, 3-a, 4-d, 5-c

Завдання для самостійної роботи

1. Задача про рюкзак 0/1 методом B&B. Реалізуйте розв’язок задачі про рюкзак 0/1 за допомогою методу гілок та меж. Розробіть функцію верхньої межі (для задачі максимізації).
2. Порівняння B&B та ДП для рюкзака. Порівняйте складність та вимоги до пам’яті для розв’язків задачі про рюкзак методами ДП та B&B. Коли один метод може бути кращим за інший?
3. Інша функція оцінки для TSP. Розробіть та опишіть альтернативну функцію нижньої межі для задачі комівояжера (наприклад, на основі мінімального кістякового дерева).
4. Стратегії пошуку. Модифікуйте вашу реалізацію TSP так, щоб вона використовувала стратегію пошуку в глибину (LIFO) замість черги з пріоритетами. Порівняйте кількість згенерованих вузлів.
5. Задача про призначення (Assignment Problem). Є n працівників та n завдань. Відома вартість виконання кожного завдання кожним працівником. Потрібно розподілити завдання так, щоб кожен виконував рівно одне, а сумарна вартість була мінімальною. Сформулюйте цю задачу для методу B&B.
6. Задача про розкрій (Cutting Stock Problem). Дослідіть цю задачу та поясніть, як метод гілок та меж може бути застосований для її розв’язання.

7. Якість межі. Поясніть на прикладі, як «туга» (точна) нижня межа дозволяє відсікти більше гілок, ніж «слабка» (дуже занижена) межа.
8. Цілочисельне програмування. Дослідіть, як метод B&B використовується для розв’язання задач цілочисельного лінійного програмування. Що є вузлом дерева пошуку в цьому випадку?
9. B&B для ігор. Чи можна застосувати метод гілок та меж до ігрових дерев (наприклад, в шахах)? Яку роль тут відіграє оцінкова функція позиції? Порівняйте з алгоритмом альфа-бета відсічення.
10. Найближчий сусід як перший «рекорд». Подумайте, як можна покращити початкове значення `best_solution_cost` в алгоритмі для TSP, щоб почати відсікати гілки раніше. (Підказка: використайте швидку евристику, наприклад, алгоритм найближчого сусіда).
11. Задача про вершинне покриття (Vertex Cover). Сформулюйте задачу пошуку мінімального вершинного покриття для методу гілок та меж. Запропонуйте можливу функцію нижньої межі.
12. Візуалізація дерева пошуку. Модифікуйте одну з ваших реалізацій (TSP або рюкзак) так, щоб вона візуалізувала або виводила в текстовий файл структуру дерева пошуку, що було згенеровано, позначаючи відсічені гілки.
13. Паралельний B&B. Дослідіть та опишіть, як метод гілок та меж можна розпаралелити для виконання на багатоядерних системах.
14. B&B для нецілочисельних задач. Чи застосовний метод B&B для задач неперервної оптимізації? Якщо так, то як? Якщо ні, то чому?
15. Есе-аналіз. Напишіть есе на тему «Мистецтво створення хорошої функції оцінки меж: компроміс між точністю та швидкістю обчислення».

Частина IV: Алгоритми на графах

Графи: Фундаментальна структура даних

1. Вступ до Теорії Графів

Графи є однією з найфундаментальніших і найуніверсальніших абстрактних комбінаторних структур у дискретній математиці та комп'ютерних науках. Вони слугують потужним інструментом для моделювання складних взаємозв'язків між об'єктами в різноманітних системах. Розуміння графів та їхніх властивостей є критично важливим для розробки ефективних алгоритмів, що лежать в основі багатьох сучасних технологій.

Визначення графа: вершини та ребра

Граф G формально визначається як упорядкована пара $G=(V,E)$, що складається з двох множин: V та E. Множина V представляє собою довільну (часто скінченну) множину вершин, які також називають вузлами або точками.

Ці вершини можуть символізувати будь-які об'єкти або сутності в моделюваній системі. Множина E складається з ребер, які також відомі як дуги, зв'язки або лінії. Кожне ребро встановлює відношення або зв'язок між двома вершинами з множини V.

Таким чином, графи дозволяють абстрагувати та візуалізувати складні мережі взаємодій, де вершини є учасниками, а ребра – їхніми зв'язками.

Основні типи графів

Залежно від характеру зв'язків та наявності додаткових атрибутів, графи поділяються на кілька основних типів, кожен з яких підходить для моделювання певних видів взаємодій.

- **Неорієнтовані графи:** У неорієнтованих графах ребра симетрично з'єднують дві вершини, що вказує на двосторонній зв'язок. Кожне ребро є невпорядкованою парою вершин, наприклад, $\{u,v\}$. Це означає, що якщо вершина u пов'язана з вершиною v, то вершина v також пов'язана з вершиною u. Прикладом може бути дружба в соціальній мережі, де зв'язок є взаємним.

- **Орієнтовані графи (діграфи):** На відміну від неорієнтованих, у орієнтованих графах ребра мають певну орієнтацію, тобто зв'язок є

асиметричним.2 Кожне ребро є впорядкованою парою вершин, наприклад, (u,v), що вказує на зв'язок від u до v, але не обов'язково у зворотному напрямку. Це може моделювати односторонні вулиці, посилання з однієї веб-сторінки на іншу або потік даних у мережі.

- **Зважені графи:** Це розширення базового визначення графа, де кожному ребру присвоєно числове значення, відоме як "вага". Вага може представляти різні метрики, такі як вартість, відстань, час, пропускна здатність або ймовірність. Ваги можуть бути як додатними, так і від'ємними.

  Однак, наявність від'ємних ваг, особливо у поєднанні з від'ємними циклами, може суттєво ускладнити застосування деяких алгоритмів пошуку найкоротших шляхів. Наприклад, класичний алгоритм Дейкстри призначений для графів з невід'ємними вагами, тоді як алгоритм Беллмана-Форда може обробляти від'ємні ваги. Зважені графи є ключовими в задачах оптимізації, де метою є мінімізація або максимізація певної величини, наприклад, пошук найдешевшого маршруту або найшвидшого часу доставки.

Розуміння основних понять теорії графів є важливим для подальшого вивчення алгоритмів.

- **Повний граф:** Граф з N вершинами (де $N > 1$), у якому кожна унікальна пара вершин з'єднана рівно одним ребром. Це означає, що в такому графі існує максимальна кількість можливих ребер.
- **Зв'язний граф:** Граф, у якому існує шлях між будь-якими двома його вершинами. Це означає, що граф складається з одного компонента зв'язності, і до будь-якої вершини можна дістатися з будь-якої іншої.
- **Дерево:** Це особливий тип зв'язного графа, який є ациклічним, тобто не містить жодних циклів. Дерева є одними з найпростіших зв'язних графів і формують основу для цілого класу задач у теорії алгоритмів, таких як пошук, сортування та організація ієрархічних даних.
- **Обхід графа:** Це систематична процедура відвідування всіх вершин та/або ребер графа. Метою обходу може бути пошук конкретної вершини, перевірка зв'язності, або виконання операцій над кожною вершиною. Найбільш відомими та широко використовуваними

методами обходу є пошук у глибину (DFS) та пошук у ширину (BFS). Ці алгоритми є фундаментальними для багатьох інших графових алгоритмів.

Застосування графів у реальному світі

Графи є невід'ємною частиною сучасної технології та науки, знаходячи застосування в широкому спектрі галузей:

- **Соціальні мережі:** Платформи, такі як $Facebook$, $LinkedIn$ та $Twitter$, інтенсивно використовують графи для моделювання взаємозв'язків між людьми, групами або організаціями. Користувачі є вершинами, а їхні дружні зв'язки, підписки чи взаємодії – ребрами. Алгоритми на графах допомагають у рекомендаціях друзів, виявленні спільнот та аналізі впливу користувачів у мережі.

- **Навігація та картографічні сервіси:** Компанії, такі як $Google\ Maps$ та $Waze$, інтенсивно використовують графи для планування маршрутів, пошуку найкоротших або найшвидших шляхів між точками та оптимізації трафіку в реальному часі. Міста або перехрестя представляються вершинами, а дороги – ребрами, часто зваженими за відстанню, часом або завантаженістю.

- **Системи рекомендацій:** Платформи електронної комерції (наприклад, $Amazon$) та стрімінгові сервіси (наприклад, $Netflix$) застосовують графові алгоритми для рекомендації продуктів, фільмів або музики користувачам. Графи моделюють взаємодію користувачів з об'єктами (наприклад, перегляди, покупки), і алгоритми знаходять схожих користувачів або об'єкти для формування персоналізованих рекомендацій.

- **Біоінформатика та геноміка:** У цій галузі графи використовуються для аналізу складних біологічних систем, таких як мережі взаємодії білків, генні регуляторні мережі та для вирівнювання послідовностей ДНК.7 Вершини можуть представляти гени, білки або метаболіти, а ребра – їхні взаємодії.

- **Комп'ютерні мережі:** Інтернет сам по собі є величезним графом, де комп'ютери та маршрутизатори є вершинами, а мережеві з'єднання – ребрами. Графи моделюють топологію мережі, а алгоритми на

графах лежать в основі маршрутних протоколів (наприклад, OSPF, BGP), забезпечуючи ефективну передачу даних та стійкість мережі.

- **Фінансові системи:** Графові алгоритми застосовуються для виявлення шахрайства, наприклад, шляхом ідентифікації підозрілих патернів у транзакційних мережах, таких як цикли відмивання грошей. Вони також використовуються для оцінки кредитоспроможності, аналізу ризиків та моделювання фінансових взаємозв'язків.
- **Управління ланцюгами поставок:** Графи допомагають аналізувати складні мережі поставок, ідентифікувати критичних постачальників, вузькі місця та вразливості. Це дозволяє оптимізувати рівні запасів, знижувати витрати та забезпечувати своєчасну доставку товарів.
- **Епідеміологія:** Для моделювання поширення хвороб у популяціях, де вершини представляють людей, а ребра – контакти або взаємодії, що можуть сприяти поширенню хвороби. Цей підхід допомагає розробляти стратегії боротьби з хворобами, планувати імунізаційні кампанії та ідентифікувати групи ризику.

Графи є не просто однією з багатьох структур даних, а фундаментальним, універсальним інструментом для абстрагування та аналізу взаємодій у будь-якій складній системі. Ця універсальність є основною причиною, чому теорія графів є однією з головних областей вивчення в дискретній математиці та комп'ютерних науках.

Здатність моделювати реальний світ у термінах вершин і ребер дозволяє застосовувати потужні алгоритмічні підходи для вирішення широкого спектра прикладних проблем. Це підкреслює, що розділ про графи є не просто теоретичним вступом, а ключовим мостом між абстрактними концепціями та їх практичним застосуванням.

Вибір конкретного типу графа для моделювання реальної задачі не є довільним, а є свідомим рішенням, яке відображає природу відношень у системі та цілі її аналізу. Наприклад, односторонні вулиці в дорожній мережі вимагають орієнтованого графа, а час проїзду по ним – зваженого. Цей зв'язок показує, що правильне моделювання є першим кроком до ефективного розв'язання проблеми. Це також вказує на те, що складність алгоритмів може залежати від властивостей графа, таких як наявність від'ємних ваг, що вимагає використання специфічних алгоритмів.

2. Історичний Екскурс: Народження

Теорія графів, як формальна математична дисципліна, вважається відносно молодою областю дискретної математики. Її витоки датуються XVIII століттям, а її народження тісно пов'язане з іменем видатного швейцарського математика Леонарда Ейлера.

Задача про кенігсберзькі мости та її значення

Першою працею, що заклала основи теорії графів, є стаття Леонарда Ейлера, опублікована у 1741 році. Вона стала результатом його роботи над знаменитою задачею про кенігсберзькі мости. Ейлер представив своє рішення Санкт-Петербурзькій академії 26 серпня 1735 року.

Суть задачі полягала в наступному: мешканці міста Кенігсберг, розташованого на річці Преголя, яка мала сім мостів, цікавилися, чи можна здійснити прогулянку містом таким чином, щоб перетнути кожен із цих мостів рівно один раз і повернутися додому. Ейлер, живучи в Кенігсберзі, сам зіткнувся з цим питанням під час своїх прогулянок.

Ключовим кроком Ейлера було його геніальне усвідомлення того, що для вирішення проблеми не має значення конкретна геометрія мостів чи ділянок суші, а лише їхні взаємозв'язки. Він "реформулював проблему в абстрактних термінах" , відкинувши несуттєві географічні деталі.

Ейлер замінив кожну ділянку суші (острови та береги річки) на абстрактну "вершину" (або вузол), а кожен міст, що їх з'єднує, – на "ребро" (зв'язок). Таким чином, він створив перше графічне представлення задачі, яке стало прототипом сучасного графа. Це був перший випадок, коли проблема була перетворена з просторової (геометрія положення) на комбінаторну.

Ейлер зробив ключове спостереження: для того, щоб пройти кожен міст рівно один раз, у будь-якій проміжній ділянці суші (вершині), кількість мостів, що її торкаються (так званий "степінь" вершини), має бути парною. Це пояснюється тим, що кожен раз, коли ви входите у вершину по мосту, ви повинні залишити її по іншому мосту.

У Кенігсберзькій задачі всі чотири ділянки суші мали непарну кількість мостів (одна – 5 мостів, три інші – по 3 мости). Оскільки шлях може мати максимум дві вершини з непарним степенем (початок і кінець), наявність чотирьох таких вершин призвела до суперечності. Отже, Ейлер дійшов висновку, що такий обхід неможливий.

У теорії графів, Ейлер показав, що існування Ейлерового шляху (обходу, що проходить кожне ребро рівно один раз) залежить від степенів вузлів. Необхідною умовою є те, що граф має бути зв'язним і мати або нуль, або рівно два вузли непарного степеня. Ця умова, як пізніше довів Карл Гірхольцер, також є достатньою. Якщо всі вершини мають парний степінь, існує Ейлерів цикл (починається і закінчується в одній точці). Робота Ейлера була опублікована під назвою "*Solutio problematis ad geometriam situs pertinentis*" (Рішення проблеми, що стосується геометрії положення).

Наступний значний імпульс теорія графів отримала приблизно через століття, коли її застосування поширилося на дослідження електричних мереж, кристалографії та органічної хімії. Це свідчить про перехід від чисто математичної цікавості до практичного інструменту в різних наукових дисциплінах.

3. Методи Представлення Графів у Пам'яті Комп'ютера

Граф, як абстрактна математична структура, потребує конкретного способу зберігання та маніпулювання в пам'яті комп'ютера. Цей спосіб називається представленням графа. Вибір відповідного представлення є вирішальним аспектом дизайну алгоритмів, оскільки він безпосередньо та суттєво впливає на ефективність та масштабованість графових алгоритмів. Різні методи представлення мають різні компроміси між використанням пам'яті (просторовою складністю) та швидкістю виконання операцій (часовою складністю).

Існують три основні техніки представлення графів:

- матриці суміжності,
- списки суміжності
- списки ребер.

Хоча список ребер не є частиною детального запиту, його згадка підкреслює різноманітність підходів. Кожен з цих методів має свої унікальні переваги та недоліки, що робить його більш або менш придатним залежно від конкретних властивостей графа (наприклад, його щільності) та типів операцій, які будуть найчастіше виконуватися.

Вибір представлення графа є прикладом одного з найважливіших принципів у комп'ютерних науках: принципу компромісу "час-пам'ять". Дослідження неодноразово підкреслюють, що вибір представлення графа

безпосередньо впливає на його ефективність та масштабованість, і що різні методи мають різні компроміси. Наприклад, матриця суміжності пропонує швидкий доступ до ребра, але використовує багато пам'яті для розріджених графів, тоді як списки суміжності економлять пам'ять, але можуть бути повільнішими для перевірки наявності ребра.

Це є фундаментальним уроком, що навчає аналітичному мисленню та прийняттю обґрунтованих рішень, виходячи з конкретних вимог задачі та доступних ресурсів. Це також показує, що "оптимальність" є контекстно-залежним поняттям.

Крім того, динамічна природа графів визначає вимоги до їх представлення. Дослідження обговорюють не лише статичні властивості графа (кількість вершин/ребер, щільність), а й динамічні операції, такі як додавання або видалення вершин/ребер.

Наприклад, додавання вершини до матриці суміжності є дуже дорогим $O(V^2)$, тоді як для списку суміжності – швидким $O(1)$. Це означає, що вибір представлення графа залежить не тільки від його поточної структури, а й від того, як часто ця структура буде змінюватися.

Якщо граф є статичним або змінюється рідко (наприклад, дорожня мережа міста), можна обрати представлення, оптимізоване для запитів. Якщо ж граф є високодинамічним (наприклад, соціальна мережа, де користувачі постійно додають/видаляють зв'язки), то перевагу слід віддати представленням, що забезпечують ефективні модифікації, навіть якщо це призведе до деяких компромісів у швидкості запитів.

4. Матриця Суміжності

Матриця суміжності є одним з найпростіших та інтуїтивно зрозумілих способів представлення графа. Вона дозволяє швидко перевіряти наявність зв'язків між вершинами, але може бути неефективною з точки зору пам'яті для певних типів графів.

Визначення та структура

Матриця суміжності – це квадратна матриця розмірністю $N \times N$, де N позначає кількість вершин у графі. Ця матриця однозначно відображає структуру графа. Кожен елемент матриці A_{ij} представляє зв'язок між вершиною i та вершиною j.

Для незваженого графа: $A_{ij} = 1$, якщо існує ребро (дуга) від вершини i до вершини j; $A_{ij} = 0$, якщо ребра немає.

Для зваженого графа: A_{ij} може зберігати вагу ребра з вершини i у вершину j. Якщо $i = j$, A_{ii} може представляти вагу самої вершини, якщо це необхідно.

Важливо відзначити, що для неорієнтованого графа матриця суміжності завжди є симетричною, тобто $A_{ij} = A_{ji}$.

Приклад представлення

Розглянемо неорієнтований граф з 4 вершинами (0, 1, 2, 3) та ребрами: (0,1), (0,2), (1,2), (2,3).
Його матриця суміжності виглядатиме так:

	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

Для орієнтованого графа з тими ж вершинами та ребрами:

$(0 \to 1)$, $(0 \to 2)$, $(1 \to 2)$, $(2 \to 3)$.
Його матриця суміжності буде такою:

	0	1	2	3
0	0	1	1	0
1	0	0	1	0
2	0	0	0	1
3	0	0	0	0

Просторова складність

Матриця суміжності вимагає $V \times V$ комірок пам'яті, де V – кількість вершин. Отже, просторова складність становить $O(V^2)$. Ця структура є ефективною з точки зору пам'яті для щільних графів, де кількість ребер E близька до максимальної V^2. Однак, для розріджених графів (де $E \ll V2$), більшість елементів матриці будуть нулями, що призводить до значного марнотратства пам'яті.

Часова складність основних операцій

- **Перевірка наявності ребра (i, j):** Ця операція є дуже швидкою, оскільки достатньо просто звернутися до елемента A_{ij} у матриці. Часова складність: $O(1)$.

- **Додавання ребра (i, j):** Щоб додати ребро, потрібно просто встановити значення $A_{ij} = 1$ (або відповідну вагу). Часова складність: $O(1)$.
- **Видалення ребра (i, j):** Аналогічно, видалення ребра полягає у встановленні $A_{ij} = 0$. Часова складність: $O(1)$.
- **Додавання вершини:** Ця операція є дуже дорогою. Вона вимагає створення нової матриці розміром $(|V|+1) \times (|V|+1)$ та копіювання всіх елементів зі старої матриці до нової. Часова складність: $O(|V|^2)$.
- **Видалення вершини:** Аналогічно додаванню, видалення вершини вимагає перебудови матриці, що включає видалення відповідного рядка та стовпця, а також зсув інших елементів або повне копіювання. Часова складність: $O(|V|^2)$.
- **Знаходження всіх сусідів вершини:** Щоб знайти всіх сусідів вершини i, необхідно просканувати весь i-й рядок матриці. У найгіршому випадку, коли вершина з'єднана з усіма іншими, це займе $O(|V|)$ часу.

Переваги та недоліки

- **Переваги:**
  - **Швидка перевірка наявності ребра:** $O(1)$ час для визначення, чи існує ребро між двома заданими вершинами.
  - **Простота реалізації:** Концептуально проста для розуміння та реалізації.
  - **Ефективність для щільних графів:** Ідеально підходить для графів, де кількість ребер близька до максимальної ($E \approx V^2$), оскільки більшість комірок матриці будуть заповнені.
  - **Краща кеш-продуктивність:** Через послідовний доступ до пам'яті, матриці суміжності можуть демонструвати кращу продуктивність кешу процесора порівняно зі списками суміжності.
- **Недоліки:**

- **Високе споживання пам’яті:** Основний недолік – $O(|V|^2)$ просторова складність, що робить її неефективною для великих і розріджених графів, де більшість пам’яті витрачається на зберігання нулів.
- **Повільні операції додавання/видалення вершин:** $O(|V|^2)$ складність для цих операцій робить її непридатною для динамічних графів, що часто змінюють свою структуру.
- **Повільний обхід сусідів:** Для алгоритмів, що вимагають ітерації по всіх сусідах вершини (наприклад, BFS, DFS), потрібно просканувати весь рядок, що є менш ефективним, ніж у списках суміжності для розріджених графів.

Фактори, що впливають на вибір матриці суміжності

Вибір матриці суміжності як представлення графа тісно пов’язаний з його щільністю. Дослідження чітко вказують, що матриця суміжності є менш ефективною для розріджених графів і марнотратно використовує пам’ять у таких випадках, тоді як для щільних графів вона підходить.

Це підкреслює, що оптимальність представлення не є абсолютною, а залежить від контексту. Неправильний вибір для розрідженого графа може призвести до значних проблем з використанням пам’яті, що є критичним для великих наборів даних. Це також вказує на необхідність інших представлень, які краще справляються з розрідженістю.

Крім того, характер найчастіше виконуваних операцій є ключовим фактором при виборі представлення. Матриця суміжності забезпечує $O(1)$ час для перевірки наявності ребра, але

$O(|V|)$ для знаходження всіх сусідів. Це демонструє, що різні операції мають різну ефективність у цьому представленні. Якщо алгоритм часто перевіряє наявність конкретного ребра (наприклад, у деяких алгоритмах, що базуються на перевірці зв’язності), матриця суміжності може бути кращою. Однак, якщо алгоритм часто обходить усіх сусідів вершини (як BFS/DFS), то матриця суміжності буде менш ефективною, оскільки їй доведеться сканувати весь рядок. Це вказує на те, що архітектура алгоритму та його взаємодія з даними мають бути узгоджені з обраним представленням графа для досягнення максимальної продуктивності.

5. Списки Суміжності

Списки суміжності є одним з найпоширеніших та гнучких способів представлення графів, особливо для розріджених графів. Вони забезпечують ефективне використання пам'яті та швидкий доступ до сусідів вершини.

Визначення та структура

Списки суміжності представляють граф у вигляді масиву, де кожен елемент масиву відповідає одній вершині графа. Кожен елемент масиву є початком списку (наприклад, зв'язаного списку або динамічного масиву), який містить усі вершини, суміжні з відповідною вершиною.

Для зважених графів, окрім ідентифікатора суміжної вершини, у списку також може зберігатися вага ребра, що з'єднує ці вершини.

Для орієнтованого графа, список для вершини u міститиме лише ті вершини v, до яких існує ребро $u \to v$. Для неорієнтованого графа, якщо існує ребро між u та v, то v буде у списку u, а u – у списку v.

Приклад представлення

Розглянемо той самий неорієнтований граф з 4 вершинами (0, 1, 2, 3) та ребрами: (0,1), (0,2), (1,2), (2,3).
Його списки суміжності виглядатимуть так:

0:
1:
2:
3:

Для орієнтованого графа з тими ж вершинами та ребрами: $(0 \to 1)$, $(0 \to 2)$, $(1 \to 2)$, $(2 \to 3)$.
Його списки суміжності такі:

0:
1:
2:
3:

Просторова складність

Просторова складність списків суміжності становить $O(|V| + |E|)$, де $|V|$ – кількість вершин, а $|E|$ – кількість ребер. Це пояснюється тим, що для кожної вершини зберігається покажчик на її список ($O(|V|)$), а сумарна довжина всіх списків дорівнює кількості ребер (для орієнтованого графа) або подвоєній кількості ребер (для неорієнтованого графа), що дає $O(|E|)$.

Ця структура є значно ефективнішою з точки зору пам'яті для розріджених графів порівняно з матрицею суміжності.

Позначимо через $deg(i)$ - степінь вершини i.

Часова складність основних операцій

- **Перевірка наявності ребра (i, j):** Щоб перевірити, чи існує ребро між вершинами i та j, необхідно просканувати список суміжності вершини i. У найгіршому випадку, це займе час, пропорційний степеню вершини i, тобто $O(deg(i))$.
- **Додавання ребра (i, j):** Додавання ребра є швидкою операцією, оскільки достатньо додати вершину j до списку вершини i (і i до списку j для неорієнтованого графа). Часова складність: $O(1)$.
- **Видалення ребра (i, j):** Видалення ребра вимагає пошуку вершини j у списку суміжності вершини i, що займає $O(deg(i))$ часу.
- **Додавання вершини:** Ця операція є дуже швидкою, оскільки достатньо просто створити новий порожній список для нової вершини. Часова складність: $O(1)$.
- **Видалення вершини:** Видалення вершини є складнішою операцією. Необхідно видалити список самої вершини, а також знайти та видалити всі входження цієї вершини зі списків інших вершин, до яких вона була суміжна. У найгіршому випадку, це може зайняти $O(|V| + |E|)$ часу.
- **Знаходження всіх сусідів вершини:** Щоб знайти всіх сусідів вершини i, необхідно просто проітеруватися по її списку суміжності. Часова складність: $O(deg(i))$.

Переваги та недоліки

- **Переваги:**
  - **Ефективне використання пам'яті:** $O(|V| + |E|)$ просторова складність робить її ідеальною для розріджених графів, де $E \ll V2$.
  - **Ефективний обхід сусідів:** Швидкий доступ до всіх сусідів вершини $O(deg(i))$, що робить її оптимальною для алгоритмів обходу, таких як BFS та DFS.

- **Швидкі операції додавання/видалення вершин:** $O(1)$ для додавання вершини, що робить її придатною для динамічних графів.

– **Недоліки:**

- **Повільніша перевірка наявності ребра:** $O(deg(i))$ час порівняно з $O(1)$ у матриці суміжності.
- **Менш кеш-дружня:** Через нерівномірне розташування даних у пам'яті (порівняно з матрицею) може мати гіршу кеш-продуктивність для деяких операцій.

Оптимальність для розріджених графів та алгоритмів обходу

Списки суміжності є найбільш ефективним представленням для розріджених графів через їх просторову ефективність, оскільки вони зберігають лише існуючі ребра, уникаючи марнотратства пам'яті на нульові елементи, як це відбувається в матриці суміжності.

Крім того, списки суміжності надають значні переваги для алгоритмів обходу, таких як пошук у ширину (BFS) та пошук у глибину (DFS). Ці алгоритми часто потребують ітерації по всіх сусідах поточної вершини. У представленні списками суміжності, це досягається за час, пропорційний кількості сусідів $O(deg(i))$. Для порівняння, у матриці суміжності для цього потрібно просканувати весь рядок, що займає

$O(|\ V\ |)$ часу. Ця різниця в ефективності робить списки суміжності кращим вибором для більшості алгоритмів, що оперують з сусідами вершин, особливо для розріджених графів. Наприклад, алгоритм Дейкстри має кращу часову складність при використанні списків суміжності $O(ElogV)$ з міні-купою порівняно з матрицею суміжності ($O(V^2)). Аналогічно, алгоритм Прима також виграє від використання списків суміжності, досягаючи складності $O((V + E)logV)$.

6. Формат стисненого рядка (CSR)

Формат стисненого рядка (Compressed Sparse Row, CSR), також відомий як формат Yale, є спеціалізованим методом представлення розріджених матриць, який знаходить широке застосування для зберігання великих розріджених графів. Він розроблений для максимальної економії пам'яті та оптимізації певних операцій, таких як множення матриці на вектор.

Визначення та структура (три одновимірні масиви)

Формат CSR представляє розріджену матрицю M (розміром $m \times n$) за допомогою трьох одновимірних масивів. Цей підхід дозволяє зберігати лише ненульові елементи, значно зменшуючи обсяг необхідної пам'яті порівняно з традиційною матрицею суміжності для розріджених графів.

1. **Масив значень (A):** Цей масив зберігає всі ненульові значення матриці (або ваги ребер для зважених графів) у порядку їх появи при порядковому обході матриці. Його розмір дорівнює загальній кількості ненульових елементів (NNZ) у матриці.
2. **Масив стовпцевих індексів (JA):** Для кожного значення в масиві A цей масив зберігає відповідний стовпцевий індекс. Його розмір також дорівнює NNZ.
3. **Масив покажчиків рядків (IA):** Цей масив зберігає кумулятивну кількість ненульових елементів до (але не включаючи) i-го рядка. Його розмір становить $m + 1$, де m – кількість рядків у матриці (або вершин у графі). Значення $IA[i]$ вказує на початковий індекс i-го рядка в масивах A та JA. Кількість ненульових елементів у рядку i можна знайти як $|IA[i + 1] - IA[i]|$.

Приклад представлення

Розглянемо приклад розрідженої матриці, що представляє граф:

0 0 0 0
5 8 0 0
0 0 3 0
0 6 0 0

Для цієї матриці:

- IA = 0 (за визначенням)
- $IA = IA +$ (кількістьненульовихурядку0) $= 0 + 0 = 0$
- $IA = IA +$ (кількістьненульовихурядку1) $= 0 + 2 = 2$
- $IA = IA +$ (кількістьненульовихурядку2) $= 2 + 1 = 3$
- $IA = IA +$ (кількістьненульовихурядку3) $= 3 + 1 = 4$

Застосування для графів

Формат CSR ідеально підходить для представлення великих, розріджених графів, особливо орієнтованих та двонаправлених. Він неявно представляє матрицю суміжності, зберігаючи лише існуючі ребра. Цей формат є

особливо цінним для високопродуктивних обчислень та роботи з дуже великими графами, які не потребують частих змін у своїй структурі. Наприклад, він використовується в бібліотеках, таких як *Boost C++ Libraries*, для ефективного зберігання графів.

Просторова складність:

Просторова складність формату CSR становить $O(NNZ + m + 1)$, що для графа, де m – кількість вершин $|V|$, а NNZ – кількість ребер $|E|$, спрощується до $O(|V| + |E|)$. Це робить його надзвичайно пам'яті-ефективним для розріджених графів, де кількість ребер значно менша за квадрат кількості вершин.

Переваги та недоліки

- **Переваги:**
  - **Надзвичайна компактність та ефективність пам'яті:** Зберігає лише ненульові елементи, що робить його ідеальним для дуже великих і розріджених графів.
  - **Швидкі операції читання та обходу:** Забезпечує швидкий доступ до вихідних ребер вершини та ефективне виконання алгоритмів обходу (BFS) та матричних операцій ($SpMV$).
  - **Підходить для високопродуктивних обчислень:** Часто використовується в наукових обчисленнях, машинному навчанні та аналізі великих графів на суперкомп'ютерах.
- **Недоліки:**
  - **Відсутність мутабельності:** Неможливість ефективно додавати або видаляти вершини чи ребра є значним обмеженням для динамічних графів.
  - **Складна структура:** Концептуально складніший для розуміння та реалізації порівняно з матрицею або списком суміжності.

Оптимізація для великих та статичних графів

Формат CSR особливо ефективний у випадках, коли більшість зв'язків у графі відсутні. Він дозволяє уникнути зберігання великої кількості нульових елементів, які б займали значний обсяг пам'яті у традиційній матриці суміжності. Це робить його незамінним для представлення веб-

графів, соціальних мереж або біологічних мереж, де кількість зв'язків є відносно невеликою порівняно з потенційною кількістю.

Однак, ця ефективність зберігання досягається за рахунок компромісу з динамічними змінами. Хоча *CSR* забезпечує високу продуктивність для статичних графів, його нездатність легко модифікувати структуру (додавати/видаляти ребра чи вершини) обмежує його застосування у сценаріях, де граф часто змінюється. Це знову ж таки підкреслює фундаментальний принцип компромісу "час-пам'ять" у дизайні структур даних: оптимальний вибір залежить від конкретних вимог до операцій та динаміки даних.

Обходи графів

У попередніх розділах ми з вами дізнались, що таке графи, чому вони важливі та як їх можна представити у пам'яті комп'ютера.

"Обхід" графа — це процес відвідування кожної вершини та кожного ребра в графі певним упорядкованим способом. Метою такого обходу може бути пошук конкретної вершини, знаходження найкоротшого шляху, визначення зв'язності компонентів графа або виконання інших завдань, які вимагають перевірки всіх частин графа.

Існує два основні та найфундаментальніші алгоритми для обходу графів: Пошук в ширину (Breadth-First Search, BFS) та Пошук в глибину (Depth-First Search, DFS). Кожен з них має свій унікальний підхід до дослідження графа, що робить їх придатними для різних завдань. Розуміння цих алгоритмів є критично важливим, адже вони є основою для багатьох складніших алгоритмів на графах.

Пошук в ширину

Пошук в ширину, або BFS, — це алгоритм для обходу або пошуку в структурі дерева або графа. Він починає з кореневої (або довільної початкової) вершини і досліджує всі сусідні вершини на поточному рівні, перш ніж перейти до вершин наступного рівня. Це можна уявити як розширення кола: ви досліджуєте все, що знаходиться на відстані 1 крок від вас, потім все, що на відстані 2 кроки, і так далі. Такий підхід гарантує, що перша знайдена вершина буде тією, до якої веде найкоротший шлях (за кількістю ребер) від початкової вершини.

Алгоритм BFS використовує структуру даних чергу (Queue) для управління послідовністю відвідування вершин. Нагадаємо, що черга — це структура

даних, яка працює за принципом “першим прийшов — першим вийшов” (FIFO - First-In, First-Out). Це означає, що елемент, який був доданий до черги найраніше, буде першим вилучений з неї.

Процес BFS виглядає так:

1. Ініціалізація:
    - Створюється порожня черга.
    - Створюється набір (або масив булевих значень) **`visited`** для відстеження вже відвіданих вершин. Це необхідно для того, щоб уникнути зациклення у графі (особливо, якщо він містить цикли) та гарантувати, що кожна вершина буде відвідана лише один раз.
    - Вибирається початкова вершина (назвемо її *S*).
    - Початкова вершина *S* позначається як відвідана (**`visited[S] = true`**) і додається до черги.
2. Обхід:
    - Доки черга не порожня, виконуються наступні кроки:
        - Вилучається вершина *U* з початку черги. Це поточна вершина, яку ми досліджуємо.
        - Для кожної сусідньої вершини *V* (тобто вершини, до якої існує ребро з *U*):
            - Якщо вершина *V* ще не була відвідана (**`visited[V] == false`**):
                - Позначити *V* як відвідану (**`visited[V] = true`**).
                - Додати *V* до кінця черги.

Цей процес продовжується доти, доки черга не стане порожньою, що означає, що всі досяжні вершини від початкової були відвідані.

Алгоритм BFS

Для кращого розуміння, давайте представимо алгоритм BFS у псевдокоді. Припустимо, що граф представлений у вигляді списків суміжності, оскільки це є ефективним представленням для алгоритмів обходу.

Структури даних, що використовуються:

- `Graph`: наш граф, що складається з вершин та ребер.
- `AdjacencyList`: список суміжності, де `AdjacencyList[U]` — це список вершин, суміжних з `U`.
- `Queue`: черга для зберігання вершин, які потрібно відвідати.
- `visited`: булевий масив (або хеш-таблиця) розміром за кількістю вершин, де `visited[V]` є `true`, якщо вершина `V` відвідана, і `false` в іншому випадку. Ініціалізується всіма `false`.
- `start_node`: початкова вершина для обходу.

```
АЛГОРИТМ BFS(Graph, start_node):
  1. СТВОРИТИ Queue Q
  2. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ ВЕРШИН, ІНІЦ
ІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ ЯК false

  3. ПОЗНАЧИТИ visited[start_node] = true
  4. ДОДАТИ start_node В Q

  5. ПОКИ Q НЕ ПОРОЖНЯ:
      a. U = ВИДАЛИТИ_З_ПОЧАТКУ(Q)  // Вилучити вершину U з черги
      b. ВИВЕСТИ_АБО_ОБРОБИТИ(U)   // Обробити вершину U (наприклад,
вивести її ID)

      c. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U:
          i. ЯКЩО visited[V] == false:
              1. ПОЗНАЧИТИ visited[V] = true
              2. ДОДАТИ V В Q
```

Операції та їх пояснення:

- `СТВОРИТИ Queue Q`: Ця операція ініціалізує порожню чергу, яка буде використовуватись для тимчасового зберігання вершин, що очікують на відвідування.
- `СТВОРИТИ БУЛЕВИЙ МАСИВ visited ...`: Ініціалізація масиву, який дозволяє нам швидко перевіряти, чи була вершина вже

відвідана. Це запобігає нескінченним циклам у графі та повторній обробці вершин.

- `ПОЗНАЧИТИ visited[start_node] = true`: Початкова вершина позначається як відвідана, щоб ми не намагались відвідати її знову.
- `ДОДАТИ start_node В Q`: Початкова вершина додається до черги, починаючи процес обходу.
- `ПОКИ Q НЕ ПОРОЖНЯ`: Основний цикл алгоритму. Він продовжується до тих пір, поки є вершини, які потрібно дослідити.
- `U = ВИДАЛИТИ_З_ПОЧАТКУ(Q)`: Ця операція вилучає вершину з "голови" черги (FIFO). Це вершина, яку ми зараз будемо досліджувати.
- `ВИВЕСТИ_АБО_ОБРОБИТИ(U)`: Це місце, де ви виконуєте будь-яку необхідну роботу з поточною вершиною. Наприклад, ви можете вивести її ідентифікатор, записати її властивості, або виконати розрахунки.
- `ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U`: Цей цикл перебирає всі вершини, які безпосередньо з'єднані з поточною вершиною U. Для представлення за допомогою списків суміжності це означає перебір елементів у списку `AdjacencyList[U]`.
- `ЯКЩО visited[V] == false`: Перевірка, чи не відвідували ми вже цю сусідню вершину V. Якщо ні, то ми її обробляємо.
- `ПОЗНАЧИТИ visited[V] = true`: Сусідня вершина V позначається як відвідана.
- `ДОДАТИ V В Q`: Сусідня вершина V додається до черги, щоб її сусіди були досліджені пізніше, після всіх вершин на поточному рівні.

Застосування BFS

BFS є надзвичайно корисним для розв'язання задач, де потрібно знайти найкоротший шлях за кількістю ребер (незважені графи), або коли необхідно дослідити граф пошарово. Типові застосування включають:

- Знаходження найкоротшого шляху у незваженому графі: Оскільки BFS досліджує всі вершини на поточному рівні перед переходом до

наступного, він гарантує, що першим знайденим шляхом до будь-якої вершини буде найкоротший шлях (за кількістю ребер).

- Побудова дерева найкоротших шляхів: BFS може бути використаний для побудови дерева, де всі шляхи від початкової вершини до інших вершин є найкоротшими.
- Визначення зв'язності компонентів: Якщо граф не повністю зв'язний, BFS може бути запущений кілька разів з невідвіданих вершин, щоб знайти всі зв'язні компоненти.
- Пошук вузлів на заданій глибині: Оскільки BFS досліджує рівень за рівнем, він ідеально підходить для таких завдань.
- Перевірка графа на циклічність: У неорієнтованому графі наявність зворотного ребра (ребра, що веде до вже відвіданої вершини, яка не є безпосереднім батьком поточної вершини в дереві обходу BFS) вказує на наявність циклу.

Продуктивність BFS

Часова складність: Часова складність BFS залежить від обраного представлення графа.

- Списки суміжності: Кожна вершина та кожне ребро будуть відвідані (або перевірені) лише один раз. Тому часова складність дорівнює $O(V + E)$, де V — кількість вершин, а E — кількість ребер. Це дуже ефективно для розріджених графів (графів з невеликою кількістю ребер порівняно з максимально можливою).
- Матриця суміжності: Для кожної вершини U потрібно переглянути весь рядок матриці суміжності (розміром V), щоб знайти її сусідів. Це займає $O(V)$ часу для однієї вершини. Оскільки ми робимо це для кожної вершини, загальна складність становить $O(V^2)$. Це може бути неефективно для великих розріджених графів.

Просторова складність: Просторова складність BFS дорівнює $O(V)$, оскільки в черзі може одночасно знаходитися до $O(V)$ вершин (у найгіршому випадку, коли всі вершини є сусідами початкової вершини), а масив `visited` також потребує $O(V)$ пам'яті.

3. **Пошук в Глибину**

Пошук в глибину, або DFS, — це ще один фундаментальний алгоритм для обходу графа або дерева. На відміну від BFS, який досліджує рівень за рівнем, DFS починає з початкової вершини і досліджує якомога далі вздовж кожної гілки, перш ніж повертатися (виконувати "зворотний крок" або backtracking) і досліджувати інші гілки.

Це можна уявити як спуск у лабіринт: ви йдете однією доріжкою до самого кінця, і лише якщо вона веде в глухий кут, ви повертаєтеся, щоб спробувати іншу.

Алгоритм DFS використовує структуру даних стек (Stack). Нагадаємо, що стек — це структура даних, яка працює за принципом "останнім прийшов — першим вийшов" (LIFO - Last-In, First-Out). Це означає, що останній елемент, доданий до стека, буде першим вилучений з нього.

Втім, найчастіше DFS реалізується рекурсивно, і в цьому випадку стек операцій виклику функцій операційної системи виконує роль явного стека.

Процес DFS (рекурсивна реалізація) виглядає так:

1. Ініціалізація:
    - Створюється набір (або масив булевих значень) `visited` для відстеження вже відвіданих вершин, щоб уникнути зациклення та повторних обходів. Всі вершини позначаються як невідвідані.
    - Вибирається початкова вершина (назвемо її *S*).
    - Викликається допоміжна рекурсивна функція `DFS_Visit(S)`.
2. Функція `DFS_Visit(U)`:
    - Вершина *U* позначається як відвідана (`visited[U] = true`).
    - Вершина *U* обробляється (наприклад, виводиться її ідентифікатор).
    - Для кожної сусідньої вершини *V* з *U*:
        - Якщо вершина *V* ще не була відвідана (`visited[V] == false`):
            - Рекурсивно викликати `DFS_Visit(V)`.

Коли рекурсивний виклик `DFS_Visit(V)` завершується (тобто всі вершини, досяжні з V, були відвідані), функція повертається до свого попереднього стану і продовжує досліджувати інших сусідів U, або повертається до вершини, що викликала U.

Алгоритм DFS

Подібно до BFS, представимо алгоритм DFS у псевдокоді. Знову ж таки, припускаємо, що граф представлений у вигляді списків суміжності.

Структури даних, що використовуються:

- `Graph`: наш граф.
- `AdjacencyList`: список суміжності.
- `visited`: булевий масив (або хеш-таблиця) розміром за кількістю вершин, ініціалізований всіма `false`.
- `start_node`: початкова вершина для обходу.

```
АЛГОРИТМ DFS(Graph, start_node):
  1. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ ВЕРШИН, ІНІЦ
ІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ ЯК false
  2. ВИКЛИКАТИ DFS_VISIT(Graph, start_node, visited)

ФУНКЦІЯ DFS_VISIT(Graph, U, visited):
  1. ПОЗНАЧИТИ visited[U] = true
  2. ВИВЕСТИ_АБО_ОБРОБИТИ(U)   // Обробка поточної вершини U

  3. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]):
      a. ЯКЩО visited[V] == false:
          i. DFS_VISIT(Graph, V, visited) // Рекурсивний виклик для н
евідвіданого сусіда
```

Операції та їх пояснення:

- `СТВОРИТИ БУЛЕВИЙ МАСИВ visited ...`: Цей масив відіграє ту ж роль, що й у BFS – він гарантує, що кожна вершина буде відвідана лише один раз.
- `ВИКЛИКАТИ DFS_VISIT(Graph, start_node, visited)`: Початковий виклик рекурсивної функції, яка починає обхід.
- `ПОЗНАЧИТИ visited[U] = true`: Вершина, яку ми щойно увійшли, позначається як відвідана.

- `ВИВЕСТИ_АБО_ОБРОБИТИ(U)`: Місце для обробки поточної вершини. Це може бути вивід, зміна стану, тощо.
- `ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U`: Ітерація по всіх сусідах поточної вершини *U*.
- `ЯКЩО visited[V] == false`: Перевірка, чи сусідня вершина *V* ще не була відвідана.
- `DFS_VISIT(Graph, V, visited)`: Ключовий крок DFS. Якщо сусід *V* не відвіданий, алгоритм рекурсивно занурюється в нього, досліджуючи цю "гілку" графа якомога глибше, перш ніж повернутися.

Застосування DFS

DFS є особливо корисним для задач, де важлива повна глибина дослідження або коли потрібно обробити елементи в певному порядку, пов'язаному з деревною структурою обходу. Типові застосування включають:

- Пошук шляхів та циклів: DFS може легко виявити, чи існує шлях між двома вершинами, або чи містить граф цикли (виявляючи зворотні ребра).
- Визначення зв'язних компонентів: Аналогічно BFS, якщо граф не повністю зв'язний, DFS може бути запущений кілька разів для знаходження всіх зв'язних компонентів.
- Топологічне сортування: Для орієнтованих ациклічних графів (DAGs), DFS може бути використаний для отримання топологічного порядку вершин (послідовності, де кожна вершина з'являється перед усіма вершинами, до яких вона має вихідні ребра).
- Знаходження сильно зв'язних компонентів: У орієнтованих графах DFS є ключовим компонентом для алгоритмів знаходження сильно зв'язних компонентів (наприклад, алгоритм Косараджу або Тар'яна).
- Вирішення лабіринтів: DFS природно підходить для таких задач, оскільки він досліджує один шлях до кінця.

Продуктивність DFS

Часова складність: Як і BFS, часова складність DFS залежить від представлення графа:

– Списки суміжності: Кожна вершина та кожне ребро будуть відвідані (або перевірені) лише один раз. Тому часова складність дорівнює $O(V + E)$, де V — кількість вершин, а E — кількість ребер. Це робить його дуже ефективним для розріджених графів.

– Матриця суміжності: Для кожної вершини U потрібно переглянути весь рядок матриці суміжності (розміром V), щоб знайти її сусідів. Це займає $O(V)$ часу для однієї вершини. Оскільки ми робимо це для кожної вершини, загальна складність становить $O(V^2)$.

Просторова складність: Просторова складність DFS дорівнює $O(V)$, оскільки у стеку рекурсивних викликів (або явному стеку, якщо використовується ітеративна реалізація) може одночасно знаходитися до $O(V)$ вершин (у найгіршому випадку, коли граф є лінійним шляхом). Масив `visited` також потребує $O(V)$ пам'яті. Важливо також враховувати глибину рекурсії, яка у найгіршому випадку може сягати V і спричинити переповнення стека для дуже великих графів, якщо мова йде про рекурсивну реалізацію.

Вибір між BFS та DFS

Вибір між BFS та DFS залежить від конкретної задачі:

– Використовуйте BFS, якщо вам потрібно знайти найкоротший шлях (за кількістю ребер) або обробити вершини рівень за рівнем.

– Використовуйте DFS, якщо вам потрібно дослідити всі можливі шляхи від певної вершини, знайти цикли, або якщо порядок обходу за глибиною є більш природним для вашої задачі (наприклад, топологічне сортування).

Обидва алгоритми є фундаментальними інструментами в арсеналі будь-якого розробника, що працює з графами, і їх глибоке розуміння відкриває двері до розв'язання широкого спектру складних задач.

Застосування Обходів Графів

У попередніх розділах ми з вами дізналися, що таке графи, чому вони важливі та як їх можна представити у пам'яті комп'ютера. Також ми докладно розглянули фундаментальні алгоритми обходу графів — пошук в

ширину (BFS) та пошук в глибину (DFS). Ці алгоритми є не просто способами відвідування вершин; вони є потужними інструментами, які, будучи застосованими з певними модифікаціями, дозволяють вирішувати широкий спектр складних задач на графах. У цьому розділі ми зануримося в деталі їхнього застосування для виявлення важливих структурних властивостей графів.

1. Компоненти зв'язності

Однією з найосновніших задач, яку можна вирішити за допомогою BFS або DFS, є знаходження компонентів зв'язності графа.

У неорієнтованому графі компонент зв'язності — це максимальний підграф, у якому між будь-якими двома вершинами існує шлях. Іншими словами, якщо ви можете дістатися з будь-якої вершини до будь-якої іншої вершини в межах цього підграфа, і цей підграф не може бути розширений шляхом додавання інших вершин графа без порушення цієї властивості, то це компонент зв'язності. Граф може складатися з однієї або декількох таких компонентів.

Наприклад, уявіть собі мережу доріг: якщо ви можете проїхати з будь-якого міста до будь-якого іншого міста в межах однієї групи міст, але не можете дістатися до міст іншої групи, то кожна така група є компонентом зв'язності.

У орієнтованому графі поняття дещо складніше:

- Слабо зв'язні компоненти: Це компоненти зв'язності базового неорієнтованого графа (тобто, якщо ігнорувати напрямки ребер).
- Сильно зв'язні компоненти (Strongly Connected Components, SCC): Це максимальні підграфи, де для будь-яких двох вершин U і V у підграфі існує шлях як від U до V, так і від V до U. Знаходження сильно зв'язних компонентів є більш складною задачею, яка зазвичай вимагає більш просунутих алгоритмів на основі DFS (наприклад, алгоритм Косараджу або Тар'яна), які виходять за рамки простого застосування DFS для пошуку компонентів зв'язності, але також базуються на принципах DFS.

Для простоти, у цьому розділі ми зосередимося на знаходженні компонентів зв'язності у неорієнтованих графах за допомогою BFS/DFS.

Ідея полягає в тому, щоб послідовно запускати BFS або DFS з кожної невідвіданої вершини графа. Кожен такий запуск виявить усі вершини, що належать до одного компонента зв'язності.

Алгоритм:

1. Ініціалізувати масив **`visited`** (або набір) для відстеження відвіданих вершин. Усі вершини спочатку позначаються як невідвідані (**`false`**). Це дозволяє нам знати, які вершини вже були частиною знайденої компоненти зв'язності.
2. Створити список (або іншу структуру) для зберігання знайдених компонентів зв'язності. Назвемо його **`components`**. Кожна компонента буде окремим списком вершин.
3. Перебрати всі вершини графа, наприклад, від 0 до $V - 1$ (де V — загальна кількість вершин).
4. Для кожної вершини i під час ітерації:
    - Якщо вершина i ще не була відвідана (**`visited[i]`** дорівнює **`false`**):
        - Це означає, що ми знайшли нову, ще не досліджену компоненту зв'язності.
        - Створити новий порожній список **`current_component`** для зберігання вершин цієї компоненти.
        - Запустити DFS (або BFS) з вершини i. Під час цього обходу:
            - Кожну відвідану вершину додавати до **`current_component`**.
            - Кожну відвідану вершину позначати як відвідану в масиві **`visited`** (**`visited[вершина] = true`**).
        - Після завершення обходу (DFS або BFS), **`current_component`** буде містити всі вершини, що належать до поточної компоненти зв'язності. Додати **`current_component`** до загального списку **`components`**.

Алгоритм знаходження компонент зв'язності (з використанням DFS)

Припустимо, що граф представлений у вигляді списків суміжності, оскільки це є ефективним представленням для алгоритмів обходу.

Структури даних, що використовуються:

- `Graph`: наш неорієнтований граф.
- `AdjacencyList`: структура, яка представляє граф. Це масив списків, де `AdjacencyList[U]` — це список вершин, суміжних з вершиною U.
- `visited`: Булевий масив розміром, рівним кількості вершин у графі. `visited[V]` є `true`, якщо вершина V вже була відвідана під час поточного або попереднього обходу, і `false` в іншому випадку. Він ініціалізується всіма `false`.
- `components`: Список списків, де кожен внутрішній список представляє вершини однієї компоненти зв'язності. Це буде кінцевий результат алгоритму.
- `current_component`: Тимчасовий список для зберігання вершин поточної компоненти, яка виявляється під час поточного виклику DFS.

```
АЛГОРИТМ FIND_CONNECTED_COMPONENTS(Graph):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Graph) // Отримуємо загальну кількість вершин у графі.
  2. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН // Ініціалізуємо масив відвіданих вершин.
  3. ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ visited ЯК false // Спочатку жодна вершина не відвідана.
  4. СТВОРИТИ ПОРОЖНІЙ СПИСОК components // Ініціалізуємо список, що зберігатиме всі знайдені компоненти.

  5. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Перебираємо всі вершини графа.
      a. ЯКЩО visited[i] == false: // Якщо вершина i ще не була відвідана (тобто не належить до вже знайденої компоненти):
          i. СТВОРИТИ ПОРОЖНІЙ СПИСОК current_component // Починаємо нову компоненту.
          ii. ВИКЛИКАТИ DFS_VISIT_FOR_COMPONENTS(Graph, i, visited, current_component) // Запускаємо DFS з цієї вершини.
          iii. ДОДАТИ current_component В components // Додаємо щойно знайдену компоненту до загального списку.

  6. ПОВЕРНУТИ components // Повертаємо список усіх компонентів зв'яз
```

```
ності.

ФУНКЦІЯ DFS_VISIT_FOR_COMPONENTS(Graph, U, visited, current_component
):
  1. ПОЗНАЧИТИ visited[U] = true // Позначаємо поточну вершину U як в
ідвідану.
  2. ДОДАТИ U В current_component // Додаємо U до списку вершин поточ
ної компоненти.

  3. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]): // Переб
ираємо всіх сусідів вершини U.
      a. ЯКЩО visited[V] == false: // Якщо сусід V ще не відвіданий:
          i. ВИКЛИКАТИ DFS_VISIT_FOR_COMPONENTS(Graph, V, visited, cu
rrent_component) // Рекурсивно викликаємо DFS для V.
```

Операції та їх пояснення:

- `КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Graph)`: Ця операція повертає загальну кількість вершин у графі. Це необхідно для правильного розміру масиву **`visited`** та циклу перебору вершин.
- `СТВОРИТИ БУЛЕВИЙ МАСИВ visited ...`: Ініціалізація масиву **`visited`** є критичною. Він діє як “пам’ять” алгоритму, дозволяючи йому уникнути повторної обробки вершин і гарантувати, що кожна вершина буде відвідана лише один раз.
- `СТВОРИТИ ПОРОЖНІЙ СПИСОК components`: Цей список буде накопичувати всі виявлені компоненти зв’язності. Кожен елемент цього списку сам по собі буде списком вершин, що належать до певної компоненти.
- `ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1)`: Цей зовнішній цикл є основним контролером. Він гарантує, що якщо граф не є повністю зв’язним (тобто, складається з декількох окремих компонентів), ми знайдемо кожну з них, починаючи новий обхід з будь-якої невідвіданої вершини.
- `ЯКЩО visited[i] == false`: Це умова, яка запускає новий обхід. Якщо вершина **`i`** ще не була відвідана, це означає, що ми натрапили на вершину, яка належить до нової, ще не виявленої компоненти зв’язності.
- `СТВОРИТИ ПОРОЖНІЙ СПИСОК current_component`: Для кожної нової компоненти ми створюємо новий порожній список. Усі

вершини, знайдені під час поточного виклику DFS, будуть додані до цього `current_component`.

- `ВИКЛИКАТИ DFS_VISIT_FOR_COMPONENTS(...)`: Цей виклик є "двигуном" для виявлення всіх вершин поточної компоненти. Він занурюється в граф з початкової вершини `i` і рекурсивно досліджує всіх її сусідів, сусідів сусідів, і так далі, доки не знайде всі досяжні вершини.
- `ДОДАТИ current_component В components`: Після того, як `DFS_VISIT_FOR_COMPONENTS` завершить виконання (тобто, всі вершини, досяжні з `i`, були знайдені та додані до `current_component`), ми додаємо цей повністю сформований список компонент до загального списку `components`.
- `ПОЗНАЧИТИ visited[U] = true`: У функції `DFS_VISIT_FOR_COMPONENTS`, коли ми вперше "входимо" у вершину U, ми одразу позначаємо її як відвідану. Це запобігає нескінченним циклам у графі та повторній обробці однієї й тієї ж вершини.
- `ДОДАТИ U В current_component`: Вершина U, яку ми щойно відвідали, додається до списку поточної компоненти.
- `ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U])`: Цей цикл перебирає всіх сусідів поточної вершини U.
- `ЯКЩО visited[V] == false`: Перед тим як рекурсивно занурюватись у сусідню вершину V, ми перевіряємо, чи вона вже була відвідана. Якщо ні, це означає, що V також належить до поточної компоненти, і ми продовжуємо обхід, викликаючи `DFS_VISIT_FOR_COMPONENTS` для V.

Цей алгоритм працює за $O(V + E)$ часу, де V — кількість вершин, а E — кількість ребер. Це пояснюється тим, що кожна вершина і кожне ребро будуть відвідані (або перевірені) лише один раз у сукупності всіх викликів DFS.

2. Мости

Мости — це критично важливі ребра у неорієнтованих графах, виявлення яких має велике значення для аналізу мережевих структур, оскільки вони представляють єдині шляхи зв'язку між частинами графа.

У неорієнтованому графі міст (або ребро-міст) — це ребро, видалення якого збільшує кількість компонентів зв'язності графа. Іншими словами, це ребро, яке є єдиним шляхом між двома компонентами графа. Якщо видалити міст, дві частини графа, які раніше були з'єднані цим мостом, стануть окремими компонентами.

Наприклад, у мережі доріг міст може представляти єдину дорогу, що з'єднує два міста (або групи міст). Якщо цю дорогу закрити, ці два міста (і всі пов'язані з ними міста) стануть недосяжними одне для одного.

Алгоритм для знаходження мостів базується на пошуку в глибину (DFS) і використовує два додаткові значення, які відстежуються для кожної вершини під час обходу:

1. **`discovery_time[U]`** (час відкриття, часто позначається як **`disc[U]`** або **`dfn[U]`**): Це час (або порядковий номер), коли вершина U була вперше відвідана під час DFS. Час починається з 0 і збільшується на 1 кожного разу, коли відкривається нова вершина.

2. **`low_link_value[U]`** (найнижче досяжне значення, часто позначається як **`low[U]`**): Це мінімальний час відкриття, досяжний з вершини U (включно з U) через ребра в дереві DFS або через одне зворотне ребро (back-edge) до вже відвіданої вершини, яка не є безпосереднім батьком U у дереві DFS.

Ідея алгоритму: Ребро (U, V) є мостом, якщо **`low_link_value[V]`** (найнижче досяжне значення для дочірньої вершини V у дереві DFS) більше, ніж **`discovery_time[U]`** (час відкриття батьківської вершини U). Це означає, що з вершини V (або будь-якої вершини в її піддереві DFS) немає зворотного шляху до U або до будь-якого предка U, окрім через ребро (U, V) саме по собі. Якщо V не може повернутись до U (або її предків), то ребро (U, V) є єдиним зв'язком між піддеревом V та рештою графа.

Алгоритм:

1. Ініціалізувати масив **`visited`** для всіх вершин як **`false`**.

2. Ініціалізувати масиви **`discovery_time`** та **`low_link_value`** для всіх вершин як нескінченність (або будь-яке дуже велике число).

3. Глобальна змінна **`time`** (або лічильник часу) ініціалізується 0. Ця змінна відстежує "поточний час" у DFS-обході.

4. Створити порожній список **bridges** для зберігання знайдених мостів.
5. Для кожної вершини i від 0 до $V - 1$:
    - Якщо вершина i ще не відвідана (**visited[i]** дорівнює **false**):
        - Викликати допоміжну функцію DFS-обходу **DFS_FIND_BRIDGES(Graph, i, -1, time, visited, discovery_time, low_link_value, bridges)**. Параметр **-1** позначає, що у початкової вершини немає батька.

Функція **DFS_FIND_BRIDGES(Graph, U, parent, time, visited, discovery_time, low_link_value, bridges)**:

1. Позначити **visited[U] = true**.
2. Збільшити глобальний **time** на 1.
3. Встановити **discovery_time[U] = time**.
4. Встановити **low_link_value[U] = time**. Спочатку припускається, що найвищим предком, якого U може досягти, є сама U.
5. Для кожної сусідньої вершини V з U (беремо зі списку суміжності **AdjacencyList[U]**):
    - Якщо V є батьківською вершиною **parent** (тобто $V == parent$):
        - ПРОПУСТИТИ це ребро. Ми не хочемо розглядати ребро, яким ми прийшли до U, як зворотне ребро, що створює цикл.
    - Якщо **visited[V] == true**:
        - Це означає, що V вже відвідана. Оскільки V не є батьком U, ребро (U, V) є зворотним ребром (back-edge) або поперечним ребром.
        - Оновити **low_link_value[U] = MIN(low_link_value[U], discovery_time[V])**. Ми оновлюємо **low_link_value[U]** з **discovery_time[V]**,

тому що U може досягти вершини V (і, можливо, її предків) через це зворотне ребро.

- Якщо `visited[V] == false`:
  - Це означає, що V ще не відвідана. Ребро (U, V) є прямим ребром у дереві DFS.
  - Рекурсивно викликати `DFS_FIND_BRIDGES(Graph, V, U, time, visited, discovery_time, low_link_value, bridges)`. Це занурюється глибше в граф.
  - Після повернення з рекурсивного виклику (тобто, після того, як все піддерево V було досліджене):
    - Оновити `low_link_value[U] = MIN(low_link_value[U], low_link_value[V])`. Це важливо, оскільки U може досягти будь-якого предка, якого може досягти V (включно з її `low_link_value`), через ребро (U, V).
    - Якщо `low_link_value[V] > discovery_time[U]`:
      - Ця умова означає, що з піддерева V немає шляху назад до U або до будь-якого предка U без використання ребра (U, V). Отже, ребро (U, V) є мостом.
      - Додати ребро (U, V) до списку `bridges`.

Алгоритм знаходження мостів (псевдокод)

```
АЛГОРИТМ FIND_BRIDGES(Graph):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Graph) // Отримуємо загальну кількість вершин.
  2. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН, ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ ЯК false // Відстежуємо відвідані вершини.
  3. СТВОРИТИ МАСИВ discovery_time РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН // Час відкриття вершини.
  4. СТВОРИТИ МАСИВ low_link_value РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН // Найнижче досяжне значення.
  5. СТВОРИТИ ПОРОЖНІЙ СПИСОК bridges // Список для зберігання знайде
```

них мостів.
  6. time = 0 // Глобальний лічильник часу для присвоєння discovery_time.

  7. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Перебираємо всі вершини графа.
      a. ЯКЩО visited[i] == false: // Якщо вершина i ще не відвідана, починаємо новий DFS-обхід.
          i. ВИКЛИКАТИ DFS_FIND_BRIDGES(Graph, i, -1, time, visited, discovery_time, low_link_value, bridges) // -1 вказує, що це корінь DFS-дерева.

  8. ПОВЕРНУТИ bridges // Повертаємо список усіх знайдених мостів.

ФУНКЦІЯ DFS_FIND_BRIDGES(Graph, U, parent, time, visited, discovery_time, low_link_value, bridges):
  1. ПОЗНАЧИТИ visited[U] = true // Позначаємо поточну вершину U як відвідану.
  2. time = time + 1 // Збільшуємо глобальний лічильник часу.
  3. discovery_time[U] = time // Встановлюємо час відкриття для U.
  4. low_link_value[U] = time   // Ініціалізуємо low_link_value для U її власним часом відкриття.

  5. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]): // Перебираємо всіх сусідів вершини U.
      a. ЯКЩО V == parent: // Якщо сусід V є батьком U в дереві DFS:
          i. ПРОДОВЖИТИ // Пропускаємо це ребро, щоб уникнути помилкового визначення циклів через ребро батько-дитина.

      b. ЯКЩО visited[V] == true: // Якщо сусід V вже відвіданий (і не є батьком):
          i. low_link_value[U] = MIN(low_link_value[U], discovery_time[V]) // Оновлюємо low_link_value U, якщо через V можна досягти предка U.
      c. ІНАКШЕ (visited[V] == false): // Якщо сусід V ще не відвіданий:
          i. ВИКЛИКАТИ DFS_FIND_BRIDGES(Graph, V, U, time, visited, discovery_time, low_link_value, bridges) // Рекурсивно викликаємо DFS для V.
          ii. low_link_value[U] = MIN(low_link_value[U], low_link_value[V]) // Оновлюємо low_link_value U, враховуючи найнижче досяжне значення з V.
          iii. ЯКЩО low_link_value[V] > discovery_time[U]: // Якщо найнижче досяжне значення з V більше, ніж час відкриття U:
              1. ДОДАТИ РЕБРО (U, V) В bridges // Це ребро є мостом.

Операції та їх пояснення:

- `discovery_time[U]`: Це **`час відкриття`** вершини U. Це значення є унікальним для кожної вершини і показує, коли саме DFS вперше "натрапив" на цю вершину. Воно встановлюється один раз при першому відвідуванні вершини.
- `low_link_value[U]`: Це **`найнижче досяжне значення`** для вершини U. Воно представляє мінімальний `discovery_time` серед усіх вершин, які досяжні з U (включно з U) через ребра в дереві DFS, а також через одне зворотне ребро (back-edge). Зворотне ребро — це ребро, що веде від вершини в піддереві DFS до її предка або до вже відвіданої вершини, яка не є її безпосереднім батьком у дереві DFS.
- `time`: Це глобальний лічильник, який збільшується при кожному новому відкритті вершини, щоб присвоїти унікальні `discovery_time`.
- `parent`: Цей параметр у рекурсивній функції `DFS_FIND_BRIDGES` відстежує батьківську вершину U у дереві DFS. Це потрібно, щоб коректно обробляти ребра: ми ігноруємо ребро, яким ми прийшли до поточної вершини, оскільки воно не є зворотним ребром, що вказує на цикл.
- `low_link_value[U] = MIN(low_link_value[U], discovery_time[V])` (у випадку `visited[V] == true`): Якщо ми зустрічаємо вже відвідану вершину V, яка не є батьком U, це означає, що ми знайшли зворотне ребро (U, V). Це зворотне ребро дозволяє U (і, отже, всьому піддереву U) "досягти" вершини V та її предків. Тому ми оновлюємо `low_link_value[U]` значенням `discovery_time[V]`, оскільки V є досяжною вершиною з меншим часом відкриття.
- `low_link_value[U] = MIN(low_link_value[U], low_link_value[V])` (після рекурсивного виклику): Після того, як ми рекурсивно дослідили піддерево, що починається з V, ми оновлюємо `low_link_value[U]`. Це означає, що U може досягти будь-якої вершини, яку може досягти V (включно з її `low_link_value`), через ребро (U, V). Таким чином, U "успадковує" найнижчий досяжний зв'язок від своїх дочірніх піддерев.

– **`ЯКЩО low_link_value[V] > discovery_time[U]`**: Це ключова умова для виявлення моста. Якщо **`low_link_value`** дочірньої вершини V (найвищий предок, якого можна досягти з V або її піддерева) є строго більшим, ніж **`discovery_time`** її батьківської вершини U, це означає, що з піддерева V немає способу повернутися до U або до будь-якого предка U без використання ребра (U, V). Отже, ребро (U, V) є мостом.

Цей алгоритм працює за $O(V + E)$ часу, що є дуже ефективним для великих графів, оскільки кожен вузол і кожне ребро обробляються лише константу разів.

3. Точки Шарніру

Точки шарніру, або точки зчленування, є ще одним важливим типом критичних вершин у неорієнтованих графах. Їхнє виявлення допомагає визначити “вузькі місця” або “точки відмови” у мережевих структурах.

У неорієнтованому графі точка шарніру (також відома як точка зчленування або розрізна вершина) — це вершина, видалення якої (разом з усіма інцидентними їй ребрами) збільшує кількість компонентів зв’язності графа. Іншими словами, це вершина, яка є “вузьким місцем” у зв’язності графа. Якщо видалити таку вершину, граф розпадається на кілька окремих частин.

Наприклад, у мережі комунікацій точка шарніру може представляти сервер, який, у разі відмови, може роз’єднати мережу на кілька ізольованих сегментів.

Алгоритм для знаходження точок шарніру також базується на Пошуку в Глибину (DFS) і використовує ті ж самі значення **`discovery_time`** та **`low_link_value`**, що й для мостів.

Ідея алгоритму: Вершина U є точкою шарніру, якщо виконується одна з наступних умов:

1. U є коренем дерева DFS, і вона має двох або більше дочірніх вершин у дереві DFS. Якщо корінь має лише одного нащадка, його видалення не роз’єднає граф (оскільки весь компонент залишиться зв’язаним через цього нащадка). Якщо ж він має двох або більше незалежних нащадків (тобто, вони знаходяться в різних піддеревах DFS, що виходять з кореня), то видалення кореня роз’єднає ці піддерева, роблячи його точкою шарніру.

2. U не є коренем дерева DFS, і існує така дочірня вершина V (у дереві DFS), що `low_link_value[V]` (найнижче досяжне значення для V або будь-якої вершини в її піддереві DFS) більше або дорівнює `discovery_time[U]` (час відкриття U). Це означає, що з піддерева V (включно з V) немає зворотного шляху до будь-якого предка U без використання самої вершини U. Таким чином, U є єдиним "шлюзом" до предків для піддерева V, і її видалення роз'єднає це піддерево від решти графа.

Алгоритм:

1. Ініціалізувати масив `visited` як `false` для всіх вершин.
2. Ініціалізувати масиви `discovery_time` та `low_link_value` для всіх вершин як нескінченність.
3. Глобальна змінна `time` ініціалізується 0.
4. Створити порожній список `articulation_points` для зберігання знайдених точок шарніру.
5. Для кожної вершини i від 0 до $V - 1$:
    - Якщо вершина i ще не відвідана:
        - Викликати `DFS_FIND_ARTICULATION_POINTS(Graph, i, -1, time, visited, discovery_time, low_link_value, articulation_points)`. (Тут `-1` позначає, що у початкової вершини немає батька).

Алгоритм знаходження точок шарніру (псевдокод)

```
АЛГОРИТМ FIND_ARTICULATION_POINTS(Graph):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Graph) // Отримуємо загальну кількість вершин.
  2. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН, ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ ЯК false // Відстежуємо відвідані вершини.
  3. СТВОРИТИ МАСИВ discovery_time РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН // Час відкриття вершини.
  4. СТВОРИТИ МАСИВ low_link_value РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН // Найнижче досяжне значення.
  5. СТВОРИТИ ПОРОЖНІЙ СПИСОК articulation_points // Список для зберігання знайдених точок шарніру.
  6. time = 0  // Глобальний лічильник часу для присвоєння discovery_time.
```

7. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Перебираємо всі вершини графа.
    a. ЯКЩО visited[i] == false: // Якщо вершина i ще не відвідана, починаємо новий DFS-обхід.
        i. ВИКЛИКАТИ DFS_FIND_ARTICULATION_POINTS(Graph, i, -1, time, visited, discovery_time, low_link_value, articulation_points) // -1 вказує, що це корінь DFS-дерева.

8. ПОВЕРНУТИ articulation_points // Повертаємо список усіх знайдених точок шарніру.

ФУНКЦІЯ DFS_FIND_ARTICULATION_POINTS(Graph, U, parent, time, visited, discovery_time, low_link_value, articulation_points):
1. ПОЗНАЧИТИ visited[U] = true // Позначаємо поточну вершину U як відвідану.
2. time = time + 1 // Збільшуємо глобальний лічильник часу.
3. discovery_time[U] = time // Встановлюємо час відкриття для U.
4. low_link_value[U] = time   // Ініціалізуємо low_link_value для U її власним часом відкриття.
5. children_count = 0 // Лічильник дочірніх вершин у дереві DFS для кореневої вершини.

6. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]): // Перебираємо всіх сусідів вершини U.
    a. ЯКЩО V == parent: // Якщо сусід V є батьком U в дереві DFS:
        i. ПРОДОВЖИТИ // Пропускаємо це ребро.

    b. ЯКЩО visited[V] == true: // Якщо сусід V вже відвіданий (і не є батьком):
        i. low_link_value[U] = MIN(low_link_value[U], discovery_time[V]) // Оновлюємо low_link_value U, якщо через V можна досягти предка U.
    c. ІНАКШЕ (visited[V] == false): // Якщо сусід V ще не відвіданий:
        i. children_count = children_count + 1 // Збільшуємо лічильник дочірніх елементів, оскільки V - це новий "прямий" нащадок.
        ii. ВИКЛИКАТИ DFS_FIND_ARTICULATION_POINTS(Graph, V, U, time, visited, discovery_time, low_link_value, articulation_points) // Рекурсивно викликаємо DFS для V.
        iii. low_link_value[U] = MIN(low_link_value[U], low_link_value[V]) // Оновлюємо low_link_value U, враховуючи найнижче досяжне значення з V.

        iv. ЯКЩО parent != -1 І low_link_value[V] >= discovery_time

```
[U]: // Умова для не-кореневих точок шарніру.
             1. ДОДАТИ U В articulation_points // U є точкою шарніру
.

  7. ЯКЩО parent == -1 І children_count >= 2: // Умова для кореневої
вершини DFS.
      a. ДОДАТИ U В articulation_points // U є точкою шарніру.
```

Операції та їх пояснення:

- `children_count`: Ця змінна використовується тільки для кореневої вершини дерева DFS (тобто, коли `parent == -1`). Вона відстежує кількість прямих нащадків (дітей) кореня. Якщо корінь має двох або більше таких нащадків (які ініціювали окремі гілки DFS), то його видалення роз'єднає ці гілки, роблячи корінь точкою шарніру. Для не-кореневих вершин ця змінна не використовується для визначення точки шарніру.
- `parent != -1`: Ця умова перевіряє, чи є поточна вершина U коренем дерева DFS. Якщо `parent` дорівнює `-1`, це означає, що U є коренем обходу DFS. Умову `parent != -1` ми використовуємо для всіх вершин, крім кореня.
- `low_link_value[V] >= discovery_time[U]`: Це основна умова для не-кореневих вершин U. Якщо найнижче досяжне значення з дочірньої вершини V (або її піддерева) є більшим або рівним часу відкриття її батька U, це означає, що всі шляхи з піддерева V до предків U проходять через U. Отже, U є критичною точкою: її видалення роз'єднає V та її піддерево від решти графа, роблячи U точкою шарніру.
- `parent == -1` І `children_count >= 2`: Ця умова стосується виключно кореневої вершини DFS. Якщо корінь має двох або більше незалежних дочірніх піддерев (тобто, з нього виходять принаймні два різні шляхи, які не повертаються один в одного), то видалення цього кореня відокремить ці піддерева одне від одного.

Алгоритм знаходження точок шарніру, як і мостів, працює за $O(V + E)$ часу, що є дуже ефективним.

4. Топологічне Сортування

Топологічне сортування — це операція, яка застосовується виключно до орієнтованих ациклічних графів (Directed Acyclic Graphs, DAGs). DAG — це

орієнтований граф, який не містить циклів. Якщо граф містить цикл, його неможливо топологічно відсортувати, оскільки існувала б нескінченна залежність.

Топологічне сортування орієнтованого ациклічного графа — це лінійний порядок всіх його вершин таким чином, що для кожного орієнтованого ребра (U, V) вершина U з'являється у цьому порядку раніше, ніж вершина V. Уявіть собі послідовність завдань, де деякі завдання залежать від завершення інших. Топологічне сортування дасть вам дійсний порядок виконання цих завдань, щоб усі залежності були задоволені.

Наприклад, кроки приготування страви, де "нарізати цибулю" має передувати "смажити цибулю". Або послідовність курсів в університеті, де "Дискретна математика" є передумовою для "Алгоритмів і структур даних".

Існує два основні алгоритми для топологічного сортування: алгоритм Кана (використовує BFS-подібний підхід) та алгоритм на основі DFS.

4.1. Алгоритм Кана (Kahn's Algorithm)

Алгоритм Кана використовує поняття вхідного ступеня (in-degree) вершини. Вхідний ступінь вершини — це кількість ребер, які входять у цю вершину.

Основна ідея: Почніть з усіх вершин, які не мають вхідних ребер (їхній вхідний ступінь дорівнює 0). Ці вершини можуть бути першими у топологічному порядку, оскільки вони не залежать від інших вершин. Видаліть їх з графа (логічно) та зменшіть вхідний ступінь всіх їхніх сусідів (тобто, вершин, до яких ведуть ребра з поточної вершини). Повторюйте цей процес до тих пір, поки не буде оброблено всі вершини.

Алгоритм:

1. Обчислити вхідний ступінь для кожної вершини графа. Це можна зробити, пройшовшись по всіх ребрах і для кожного ребра (U, V) збільшивши лічильник `in_degree[V]` на 1.
2. Створити чергу `Q` і додати до неї всі вершини, які мають вхідний ступінь 0. Ці вершини є початковими точками для топологічного сортування.
3. Створити порожній список `topological_order` для зберігання результату (відсортованих вершин).

4. Змінна `count` ініціалізується 0. Вона буде відстежувати кількість оброблених вершин.
5. Доки черга `Q` не порожня:
    - Вилучити вершину U з початку черги `Q`. Це поточна вершина, яку ми додаємо до нашого порядку.
    - Додати U до `topological_order`.
    - Збільшити `count` на 1.
    - Для кожної сусідньої вершини V з U (тобто, для кожного ребра (U, V)):
        - Зменшити вхідний ступінь V на 1. Це імітує видалення ребра (U, V) з графа, оскільки залежність V від U тепер задоволена.
        - Якщо вхідний ступінь V стає 0, додати V до черги `Q`. Це означає, що V тепер не має невирішених залежностей і може бути наступною у порядку.
6. Після завершення циклу:
    - Якщо `count` менше, ніж загальна кількість вершин графа, це означає, що в графі присутній цикл, і топологічне сортування неможливе. У цьому випадку алгоритм виводить повідомлення про цикл і повертає порожній список.
    - Інакше `topological_order` містить дійсний топологічний порядок.

Псевдокод алгоритму Кана

Структури даних, що використовуються:

– `Graph`: орієнтований ациклічний граф (DAG).

– `AdjacencyList`: структура, яка представляє граф. Це масив списків, де `AdjacencyList[U]` — це список вершин, суміжних з вершиною U (тобто, вершин V, до яких ведуть ребра з U).

– `in_degree`: Масив розміром, рівним кількості вершин. `in_degree[V]` зберігає кількість вхідних ребер для вершини V.

- `Queue Q`: Черга для зберігання вершин, які мають вхідний ступінь 0 і готові до обробки.
- `topological_order`: Список для зберігання відсортованих вершин. Це буде кінцевий результат.

```
АЛГОРИТМ KAHN_TOPOLOGICAL_SORT(Graph):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Graph) // Отримуємо загальну кількість вершин у графі.
  2. СТВОРИТИ МАСИВ in_degree РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН // Ініціалізуємо масив для зберігання вхідних ступенів.
  3. ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ in_degree ЯК 0 // Спочатку всі вхідні ступені нульові.

  4. // Крок 1: Обчислити вхідний ступінь для кожної вершини
  5. ДЛЯ КОЖНОЇ ВЕРШИНИ U В Graph: // Проходимо по всіх вершинах U у графі.
      a. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]): // Проходимо по всіх сусідах V вершини U.
          i. in_degree[V] = in_degree[V] + 1 // Збільшуємо вхідний ступінь для вершини V, оскільки є ребро (U, V).

  6. // Крок 2: Додати вершини з вхідним ступенем 0 до черги
  7. СТВОРИТИ Queue Q // Створюємо порожню чергу.
  8. ДЛЯ КОЖНОЇ ВЕРШИНИ і ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Перебираємо всі вершини графа.
      a. ЯКЩО in_degree[i] == 0: // Якщо вхідний ступінь вершини і дорівнює 0:
          i. ДОДАТИ і В Q // Додаємо її до черги, вона готова до обробки.

  9. // Крок 3: Виконати сортування
  10. СТВОРИТИ ПОРОЖНІЙ СПИСОК topological_order // Створюємо список для зберігання результату сортування.
  11. count = 0 // Лічильник оброблених вершин.

  12. ПОКИ Q НЕ ПОРОЖНЯ: // Доки черга не порожня, продовжуємо обробку.
      a. U = ВИДАЛИТИ_З_ПОЧАТКУ(Q) // Вилучаємо вершину U з початку черги (перша додана, перша вилучена).
      b. ДОДАТИ U В topological_order // Додаємо U до кінцевого топологічного порядку.
      c. count = count + 1 // Збільшуємо лічильник оброблених вершин.

      d. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]): // Д
```

```
ля кожного сусіда V вершини U (ребро (U, V)):
          i. in_degree[V] = in_degree[V] - 1 // Зменшуємо вхідний сту
пінь V, оскільки залежність V від U задоволена.
          ii. ЯКЩО in_degree[V] == 0: // Якщо вхідний ступінь V стає
0:
              1. ДОДАТИ V В Q // V тепер готова до обробки, додаємо ї
ї до черги.

  13. // Крок 4: Перевірка на цикли
  14. ЯКЩО count < КІЛЬКІСТЬ_ВЕРШИН: // Якщо кількість оброблених вер
шин менша за загальну кількість вершин:
      a. ВИВЕСТИ "Граф містить цикл, топологічне сортування неможливе
." // Це означає, що є цикл.
      b. ПОВЕРНУТИ ПОРОЖНІЙ_СПИСОК // У цьому випадку топологічний по
рядок не існує.
  15. ІНАКШЕ:
      a. ПОВЕРНУТИ topological_order // Повертаємо отриманий топологі
чний порядок.
```

Операції та їх пояснення:

- `in_degree`: Цей масив зберігає вхідний ступінь для кожної вершини. Вхідний ступінь вершини V — це кількість ребер (U, V), що входять у V. Його попереднє обчислення є ключовим для цього алгоритму.
- `ЯКЩО in_degree[i] == 0: ДОДАТИ i В Q`: Вершини, які мають вхідний ступінь 0, не залежать від жодних інших вершин у графі. Тому вони можуть бути першими у топологічному порядку і додаються до черги для початкової обробки.
- `U = ВИДАЛИТИ_З_ПОЧАТКУ(Q)`: Ця операція вилучає вершину з “голови” черги. Це вершина, яка вже оброблена і може бути додана до топологічного порядку, оскільки всі її залежності (якщо такі були) вже задоволені.
- `in_degree[V] = in_degree[V] - 1`: Коли ми обробляємо вершину U і додаємо її до топологічного порядку, це еквівалентно її “видаленню” з графа разом з усіма її вихідними ребрами. Тому для всіх сусідів V вершини U ми зменшуємо їхній вхідний ступінь, оскільки залежність V від U тепер задоволена.
- `ЯКЩО in_degree[V] == 0: ДОДАТИ V В Q`: Якщо вхідний ступінь сусідньої вершини V стає 0 після зменшення, це означає, що всі її

залежності тепер задоволені. Отже, V готова до обробки і додається до черги.

- `count < КІЛЬКІСТЬ_ВЕРШИН`: Ця перевірка є ключовою для виявлення циклів. Якщо після завершення алгоритму кількість оброблених вершин (`count`) менша за загальну кількість вершин у графі, це означає, що залишилися вершини, які все ще мають ненульовий вхідний ступінь і не могли бути оброблені. Це можливо лише у випадку, якщо ці вершини утворюють цикл, і тому їхній вхідний ступінь ніколи не може стати 0. Таким чином, алгоритм Кана може не тільки сортувати DAG, а й виявляти цикли.

Часова складність алгоритму Кана становить $O(V + E)$, де V — кількість вершин, а E — кількість ребер. Це пояснюється тим, що ми обчислюємо вхідні ступені всіх вершин ($O(V + E)$), а потім кожна вершина додається та вилучається з черги один раз, і кожне ребро обробляється один раз при зменшенні вхідного ступеня сусіда. Просторова складність також $O(V + E)$ для зберігання списків суміжності та `in_degree` масиву, плюс $O(V)$ для черги.

4.2. Топологічне Сортування на основі DFS

Алгоритм топологічного сортування на основі DFS використовує властивість DFS-обходу: вершина може бути додана до топологічного порядку лише після того, як всі її залежності (тобто всі вершини, до яких вона має вихідні ребра, і які знаходяться “нижче” неї у потоці залежностей) були оброблені.

Основна ідея: Виконати DFS-обхід графа. Вершини додаються до топологічного порядку після того, як всі їхні нащадки були відвідані та оброблені. Це означає, що вершина U додається до порядку лише тоді, коли DFS завершує обхід усього піддерева, що виходить з U. Якщо ви додаєте вершини до списку у порядку “завершення” (post-order), а потім перевертаєте цей список, ви отримаєте топологічний порядок.

Алгоритм:

1. Ініціалізувати масив `visited` як `false` для всіх вершин.
2. Створити порожній стек (або список) `topological_order_stack` для зберігання вершин. Стек використовується тому, що він працює за принципом LIFO (Last-In, First-Out), що дозволяє нам збирати вершини у зворотному порядку завершення.

3. Для кожної вершини i від 0 до $V - 1$:
    - Якщо вершина i ще не відвідана:
        - Викликати допоміжну функцію DFS-обходу `DFS_TOPOLOGICAL_SORT_VISIT(Graph, i, visited, topological_order_stack)`.
4. Після завершення обходу всіх компонентів, елементи в `topological_order_stack` будуть знаходитись у зворотному топологічному порядку. Вилучити їх зі стека по черзі та додати до нового списку, щоб отримати кінцевий топологічний порядок.

Псевдокод топологічного сортування на основі DFS

Структури даних, що використовуються:

- `Graph`: орієнтований ациклічний граф (DAG).
- `AdjacencyList`: список суміжності.
- `visited`: Булевий масив для відстеження відвіданих вершин.
- `topological_order_stack`: Стек для зберігання вершин у порядку завершення обходу (порядок “пост-обходу”).
- `final_topological_order`: Список для зберігання кінцевого топологічного порядку.

```
АЛГОРИТМ DFS_TOPOLOGICAL_SORT_MAIN(Graph):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Graph) // Отримуємо загальну кількість вершин.
  2. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН, ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ ЯК false // Відстежуємо відвідані вершини.
  3. СТВОРИТИ ПОРОЖНІЙ Stack topological_order_stack // Створюємо стек для результатів.

  4. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Перебираємо всі вершини графа.
      a. ЯКЩО visited[i] == false: // Якщо вершина i ще не відвідана, починаємо новий DFS-обхід.
          i. ВИКЛИКАТИ DFS_TOPOLOGICAL_SORT_VISIT(Graph, i, visited, topological_order_stack)

  5. СТВОРИТИ ПОРОЖНІЙ СПИСОК final_topological_order // Список для зберігання остаточного порядку.
```

```
  6. ПОКИ topological_order_stack НЕ ПОРОЖНІЙ: // Вилучаємо елементи зі стека, щоб отримати правильний порядок.
      a. U = ВИДАЛИТИ_З_ВЕРШИНИ(topological_order_stack) // Вилучаємо вершину з вершини стека.
      b. ДОДАТИ U В final_topological_order // Додаємо її до фінального списку.

  7. ПОВЕРНУТИ final_topological_order // Повертаємо отриманий тополо гічний порядок.

ФУНКЦІЯ DFS_TOPOLOGICAL_SORT_VISIT(Graph, U, visited, topological_ord er_stack):
  1. ПОЗНАЧИТИ visited[U] = true // Позначаємо поточну вершину U як в ідвідану.

  2. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]): // Перебираємо всіх сусідів V вершини U.
      a. ЯКЩО visited[V] == false: // Якщо сусід V ще не відвіданий:
          i. ВИКЛИКАТИ DFS_TOPOLOGICAL_SORT_VISIT(Graph, V, visited, topological_order_stack) // Рекурсивно занурюємось у V.

  3. ДОДАТИ U В topological_order_stack // Додаємо вершину U в стек п ісля відвідування всіх її нащадків.
```
Операції та їх пояснення:

- `DFS_TOPOLOGICAL_SORT_MAIN(Graph)`: Це основна функція, яка керує процесом топологічного сортування. Вона ініціалізує необхідні структури даних і запускає DFS-обхід для всіх невідвіданих вершин.
- `visited`: Цей масив використовується для відстеження відвіданих вершин, щоб уникнути нескінченних рекурсій у випадку циклів (хоча для DAGs їх не повинно бути) та повторної обробки вершин.
- `topological_order_stack`: Це стек, до якого вершини додаються після того, як всі їхні сусіди (і їхні піддерева) були повністю досліджені (тобто, після завершення рекурсивних викликів для всіх нащадків). Оскільки стек є LIFO-структурою, вершина, яка була “останньою” у гілці обходу (тобто, найглибшою), буде додана першою.
- `ЯКЩО visited[i] == false`: Зовнішній цикл гарантує, що ми обробимо всі компоненти графа, навіть якщо він не зв’язний.

– **`ДОДАТИ U В topological_order_stack`** (після циклу сусідів): Це ключовий момент цього алгоритму. Вершина U додається до стека лише після того, як всі її сусіди V (і, рекурсивно, всі їхні нащадки) були повністю відвідані та оброблені. Це гарантує, що якщо існує ребро $U \to V$, то V буде додана до стека раніше, ніж U. Коли стек потім буде вивантажений (вилучаючи елементи з вершини), U з'явиться раніше, ніж V, що відповідає визначенню топологічного сортування (передумова U перед V).

– **`ПОКИ topological_order_stack НЕ ПОРОЖНІЙ ... ДОДАТИ U В final_topological_order`**: Після того, як DFS завершив обхід всіх вершин, стек **`topological_order_stack`** містить вершини у зворотному топологічному порядку (порядок завершення DFS). Виймаючи їх зі стека по черзі, ми отримуємо правильний топологічний порядок.

Часова складність цього алгоритму також $O(V + E)$, оскільки це, по суті, модифікований DFS, де кожна вершина та кожне ребро обробляються константу разів. Просторова складність $O(V)$ для стека рекурсії (глибина стека може сягати V у найгіршому випадку, наприклад, для лінійного графа) та масиву **`visited`**.

Порівняння алгоритмів Кана та на основі DFS:

Обидва алгоритми Кана та на основі DFS виконують топологічне сортування за однакову часову та просторову складність.

– Алгоритм Кана є більш інтуїтивним для розуміння з точки зору "задоволення залежностей" і часто легше реалізується ітеративно. Він вимагає попереднього обчислення вхідних ступенів, що є додатковим початковим проходом по графу.

– Алгоритм на основі DFS є більш елегантним, якщо ви вже знайомі з рекурсивним DFS, і природно виводить порядок "після обходу". Він також може бути легко адаптований для виявлення циклів, якщо граф не є DAG (про що ми поговоримо далі).

Вибір між ними часто залежить від особистих переваг розробника або від того, чи потрібні інші властивості (наприклад, раннє виявлення циклів у алгоритмі Кана).

5. Пошук Циклів

Виявлення циклів у графі — це ще одна важлива задача, яку можна вирішити за допомогою DFS. Цикл — це шлях, який починається і закінчується в одній і тій же вершині, при цьому всі інші вершини на шляху відрізняються.

Цикли дуже часто зустрічаються:

- У топологічному сортуванні (як ми бачили вище) цикли в орієнтованих графах унеможливлюють сортування, вказуючи на логічну суперечність у залежностях.
- У розкладних системах (наприклад, управління завданнями в операційних системах) цикли можуть вказувати на взаємні блокування (deadlocks) — ситуації, коли два або більше процесів чекають один на одного в нескінченному циклі.
- У фінансових мережах або графах потоків товарів цикли можуть вказувати на можливості арбітражу (безризикового отримання прибутку) або нескінченні джерела/стоки.
- У соціальних мережах цикли формують “замкнуті” групи або “ехо-камери”, де інформація циркулює в межах обмеженого кола.
- У моделюванні біологічних систем (наприклад, метаболічних шляхів) цикли можуть вказувати на замкнуті процеси.

Підхід до пошуку циклів відрізняється для неорієнтованих та орієнтованих графів.

5.1. Пошук циклів у неорієнтованих графах (за допомогою DFS)

У неорієнтованому графі цикл існує, якщо під час DFS-обходу ми зустрічаємо вже відвідану вершину, яка не є безпосереднім батьком поточної вершини у дереві DFS.

Коли ми виконуємо DFS, ми підтримуємо масив `visited` (або набір), щоб уникнути повторної обробки вершин. Якщо ми зустрічаємо сусіда V поточної вершини U, який вже відвіданий (`visited[V] == true`), і V при цьому не є `parent` (батьківською вершиною U у дереві DFS), то це означає, що ми знайшли зворотне ребро (back-edge). Це зворотне ребро завершує цикл.

Наприклад, якщо ми йдемо від A до B, потім до C, а з C бачимо ребро до A, і A не є батьком C (батьком C є B), то ми знайшли цикл A-B-C-A.

Алгоритм:

1. Ініціалізувати масив `visited` як `false` для всіх вершин.
2. Для кожної вершини i від 0 до $V - 1$:
    - Якщо вершина i ще не відвідана (`visited[i]` дорівнює `false`):
        - Викликати допоміжну функцію DFS-обходу `DFS_DETECT_CYCLE_UNDIRECTED(Graph, i, -1, visited)`. Параметр `-1` позначає, що у початкової вершини немає батька.
        - Якщо цей виклик поверне `true` (тобто, цикл був знайдений у компоненті, що містить i), то негайно повернути `true`, оскільки ми вже знаємо, що граф містить хоча б один цикл.
3. Якщо жоден з викликів `DFS_DETECT_CYCLE_UNDIRECTED` не знайшов циклу (тобто, всі компоненти були оброблені без виявлення циклів), повернути `false`.

Псевдокод пошуку циклів у неорієнтованих графах

Структури даних, що використовуються:

– `Graph`: неорієнтований граф.

– `AdjacencyList`: список суміжності.

– `visited`: Булевий масив для відстеження відвіданих вершин.

```
АЛГОРИТМ DETECT_CYCLE_UNDIRECTED(Graph):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Graph) // Отримуємо загальну кількість вершин.
  2. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН, ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ ЯК false // Відстежуємо відвідані вершини.

  3. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Перебираємо всі вершини графа.
      a. ЯКЩО visited[i] == false: // Якщо вершина i ще не відвідана, починаємо новий DFS-обхід.
          i. ЯКЩО ВИКЛИКАТИ DFS_DETECT_CYCLE_UNDIRECTED(Graph, i, -1, visited) ПОВЕРТАЄ true: // Якщо DFS знайшов цикл:
```

```
        1. ПОВЕРНУТИ true // Негайно повертаємо true.

  4. ПОВЕРНУТИ false // Якщо жодного циклу не знайдено в жодній компо
ненті, повертаємо false.

ФУНКЦІЯ DFS_DETECT_CYCLE_UNDIRECTED(Graph, U, parent, visited):
  1. ПОЗНАЧИТИ visited[U] = true // Позначаємо поточну вершину U як в
ідвідану.

  2. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]): // Переб
ираємо всіх сусідів V вершини U.
      a. ЯКЩО V == parent: // Якщо сусід V є батьком U:
          i. ПРОДОВЖИТИ // Пропускаємо це ребро, оскільки це ребро, я
ким ми прийшли до U, і воно не формує цикл.

      b. ЯКЩО visited[V] == true: // Якщо сусід V вже відвіданий (і н
е є батьком):
          i. ПОВЕРНУТИ true // Знайдено цикл! Це зворотне ребро, яке
вказує на цикл.

      c. ІНАКШЕ (visited[V] == false): // Якщо сусід V ще не відвідан
ий:
          i. ЯКЩО ВИКЛИКАТИ DFS_DETECT_CYCLE_UNDIRECTED(Graph, V, U,
visited) ПОВЕРТАЄ true: // Рекурсивно занурюємось у V.
              1. ПОВЕРНУТИ true // Якщо рекурсивний виклик знайшов ци
кл, передаємо true далі.
  3. ПОВЕРНУТИ false // Якщо з цієї вершини U та її піддерева цикл не
знайдено, повертаємо false.
```

Операції та їх пояснення:

- `parent`: Цей параметр у рекурсивній функції є критичним. У неорієнтованому графі ребро (U, V) означає, що існує зв'язок в обох напрямках. Без відстеження батька, ребро від V до U (якщо U є батьком V) буде помилково ідентифіковано як цикл. Тому ми ігноруємо ребро, яке веде назад до безпосереднього батька.
- `ЯКЩО visited[V] == true` І `V != parent`: Це ключова умова для виявлення циклу в неорієнтованому графі. Вона спрацьовує, коли DFS натрапляє на вже відвідану вершину V, яка не є безпосереднім батьком U. Це вказує на зворотне ребро, яке створює цикл.
- `ПОВЕРНУТИ true`: Як тільки цикл виявлено, функція негайно повертає `true` до попередніх викликів, сигналізуючи про наявність циклу.

– Часова складність: $O(V + E)$, оскільки це DFS-обхід.

5.2. Пошук циклів у орієнтованих графах (за допомогою DFS)

Пошук циклів в орієнтованих графах складніший, оскільки “зворотне ребро” (як у неорієнтованих графах) не обов’язково означає цикл, якщо це ребро веде до вже відвіданої вершини, яка не є предком у поточному шляху DFS. Для орієнтованих графів цикл існує, якщо ми зустрічаємо зворотне ребро, яке веде до вершини, що вже перебуває в поточному шляху DFS (тобто, вона ще не завершила обхід свого піддерева і знаходиться у стеку рекурсії).

Щоб відстежувати, які вершини перебувають у поточному шляху DFS, ми використовуємо три стани відвідування для кожної вершини:

1. `UNVISITED` (Не відвідана): Вершина ще не була відвідана.
2. `VISITING` (Відвідується / В процесі): Вершина була відвідана, і вона наразі перебуває у стеку рекурсії DFS (тобто, DFS почав її обхід, але ще не завершив обхід всіх її нащадків).
3. `VISITED` (Відвідана / Завершено): Вершина та всі її нащадки повністю оброблені, і вона була вилучена зі стека рекурсії.

Якщо під час DFS ми зустрічаємо ребро (U, V), де V знаходиться у стані `VISITING`, це означає, що ми знайшли зворотне ребро до вершини, яка є предком U у поточному шляху DFS, що формує цикл.

Алгоритм:

1. Ініціалізувати масив `state` для всіх вершин як `UNVISITED`.
2. Для кожної вершини i від 0 до $V - 1$:
    - Якщо вершина i знаходиться у стані `UNVISITED`:
        - Викликати `DFS_DETECT_CYCLE_DIRECTED(Graph, i, state)`.
        - Якщо цей виклик поверне `true` (цикл знайдено), то негайно повернути `true`.
3. Якщо жоден з викликів не знайшов циклу, повернути `false`.

Псевдокод пошуку циклів у орієнтованих графах

Структури даних, що використовуються:

- `Graph`: орієнтований граф.
- `AdjacencyList`: список суміжності.
- `state`: Масив для відстеження стану відвідування кожної вершини (UNVISITED, VISITING, VISITED).

```
// Визначення станів вершин
КОНСТАНТА UNVISITED = 0
КОНСТАНТА VISITING = 1
КОНСТАНТА VISITED = 2

АЛГОРИТМ DETECT_CYCLE_DIRECTED(Graph):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Graph) // Отримуємо загальну кількість вершин.
  2. СТВОРИТИ МАСИВ state РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН, ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ ЯК UNVISITED // Ініціалізуємо стани вершин.

  3. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Перебираємо всі вершини графа.
      a. ЯКЩО state[i] == UNVISITED: // Якщо вершина і ще не була відвідана:
          i. ЯКЩО ВИКЛИКАТИ DFS_DETECT_CYCLE_DIRECTED(Graph, i, state) ПОВЕРТАЄ true: // Якщо DFS знайшов цикл:
              1. ПОВЕРНУТИ true // Негайно повертаємо true.

  4. ПОВЕРНУТИ false // Якщо жодного циклу не знайдено, повертаємо false.

ФУНКЦІЯ DFS_DETECT_CYCLE_DIRECTED(Graph, U, state):
  1. state[U] = VISITING // Позначаємо поточну вершину U як "в процесі відвідування".

  2. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З AdjacencyList[U]): // Перебираємо всіх сусідів V вершини U.
      a. ЯКЩО state[V] == UNVISITED: // Якщо сусід V ще не відвіданий:
          i. ЯКЩО ВИКЛИКАТИ DFS_DETECT_CYCLE_DIRECTED(Graph, V, state) ПОВЕРТАЄ true: // Рекурсивно занурюємось у V.
              1. ПОВЕРНУТИ true // Якщо рекурсивний виклик знайшов цикл, передаємо true далі.

      b. ІНАКШЕ ЯКЩО state[V] == VISITING: // Якщо сусід V зараз "в процесі відвідування" (у стеку рекурсії):
          i. ПОВЕРНУТИ true // Знайдено цикл! Це зворотне ребро до ве
```

```
ршини у поточному шляху DFS.

  3. state[U] = VISITED // Позначаємо поточну вершину U як "відвідану
" (обхід її піддерева завершено).
  4. ПОВЕРНУТИ false // Якщо з цієї вершини U та її піддерева цикл не
знайдено, повертаємо false.
```

Операції та їх пояснення:

- `state`: Це масив, що відстежує стан кожної вершини.
  - `UNVISITED`: Вершина ще не розглядалась DFS.
  - `VISITING`: Вершина була відвідана, і її обробка ще не завершена (тобто вона є частиною поточного рекурсивного шляху DFS).
  - `VISITED`: Вершина була повністю оброблена (усі її нащадки також були відвідані), і вона вже не є частиною активного рекурсивного шляху.
- `state[U] = VISITING`: Коли ми починаємо обробляти вершину U, ми позначаємо її як `VISITING`. Це критично, оскільки це дозволяє нам виявляти цикли, якщо ми знову натрапимо на U, перш ніж її обробка буде завершена.
- `ЯКЩО state[V] == UNVISITED`: Це стандартний крок DFS, коли ми рекурсивно досліджуємо нову, невідвідану гілку.
- `ІНАКШЕ ЯКЩО state[V] == VISITING`: Це ключова умова для виявлення циклу в орієнтованому графі. Якщо ми зустрічаємо сусіда V, який вже знаходиться у стані `VISITING`, це означає, що V є предком U у поточному шляху DFS. Отже, ребро (U, V) замикає цикл.
- `state[U] = VISITED`: Після того, як усі сусіди U були розглянуті (і рекурсивні виклики для них завершились), ми позначаємо U як `VISITED`. Це означає, що U повністю оброблена, і вона може бути “вилучена” з уявного стека рекурсії.

Часова складність: $O(V + E)$, оскільки це DFS-обхід, де кожна вершина та ребро обробляються константу разів. Просторова складність $O(V)$ для масиву `state` та стека рекурсії.

Контрольні Запитання

1. Пошук в ширину (BFS) проти Пошук в глибину (DFS): Поясніть основну відмінність між BFS і DFS з точки зору їхнього підходу до обходу графа. Які структури даних використовуються кожним з них для управління послідовністю відвідування вершин?

2. Компоненти Зв'язності: Що таке компонент зв'язності в неорієнтованому графі? Опишіть загальний підхід до їхнього знаходження за допомогою DFS.

3. Мости: Дайте визначення мосту (ребра-моста) у неорієнтованому графі. Які два ключові значення (`discovery_time` та `low_link_value`) використовуються для їхнього виявлення за допомогою DFS, і як вони інтерпретуються?

4. Точки шарніру: Що таке точка шарніру (точка зчленування) у неорієнтованому графі? Опишіть дві умови, за яких вершина може бути визначена як точка шарніру, розрізняючи випадок кореня дерева DFS та не-кореневої вершини.

5. Топологічне сортування: Яка особлива властивість графа є обов'язковою для застосування топологічного сортування? Чому?

6. Алгоритм Кана: Опишіть основну ідею алгоритму Кана для топологічного сортування. Який параметр вершини є ключовим для цього алгоритму? Як алгоритм Кана виявляє цикли?

7. DFS-орієнтоване топологічне сортування: Поясніть принцип топологічного сортування на основі DFS. У якому порядку вершини додаються до результату, і чому?

8. Пошук циклів у неорієнтованих графах: Як виявляються цикли у неорієнтованих графах за допомогою DFS? Яка додаткова інформація, крім `visited`, потрібна для цього?

9. Пошук циклів у орієнтованих графах: Чому пошук циклів в орієнтованих графах складніший, ніж у неорієнтованих? Які три стани відвідування використовуються для виявлення циклів в орієнтованих графах за допомогою DFS, і що означає кожен стан?

10. Застосування на практиці: Наведіть приклади реальних задач, де застосування знаходження мостів або точок шарніру є критично важливим для аналізу мереж.

Методичні вказівки до лабораторної роботи

1. Назва

   Аналіз структурних властивостей графів за допомогою обходів DFS/BFS

2. Мета

   Метою даної лабораторної роботи є набуття практичних навичок у реалізації та застосуванні алгоритмів пошуку в глибину (DFS) та пошуку в ширину (BFS) для аналізу ключових структурних властивостей графів, включаючи знаходження компонентів зв’язності, мостів, точок шарніру, топологічного сортування та виявлення циклів.

3. Завдання

a) Реалізація представлення графа: Створити клас або структуру для представлення графа у пам’яті (наприклад, за допомогою списків суміжності).

b) Реалізація базового DFS: Розробити функцію для виконання обходу графа в глибину.

c) Застосування DFS для компонент зв’язності: Реалізувати функцію, що знаходить всі компоненти зв’язності у неорієнтованому графі.

d) Застосування DFS для мостів: Реалізувати функцію, що виявляє всі мости у неорієнтованому графі.

e) Застосування DFS для точок шарніру: Реалізувати функцію, що визначає всі точки шарніру у неорієнтованому графі.

f) Топологічне сортування (DFS-based): Реалізувати алгоритм топологічного сортування для орієнтованих ациклічних графів (DAG) на основі DFS.

g) Топологічне сортування (алгоритм Кана): Реалізувати алгоритм Кана для топологічного сортування DAG.

h) Пошук циклів: Реалізувати функції для виявлення циклів у неорієнтованих та орієнтованих графах.

i) Тестування: Протестувати всі реалізовані алгоритми на наданих або власноруч створених тестових графах.

4. Опис

Вам необхідно розробити програмний модуль на обраній мові програмування (наприклад, Python, C++, Java, JavaScript). Модуль повинен дозволяти:

– Створювати графи (орієнтовані та неорієнтовані) з заданою кількістю вершин та ребер.

– Додавати ребра до графа.

– Викликати реалізовані функції для аналізу графа (компоненти зв'язності, мости, точки шарніру, топологічне сортування, пошук циклів).

– Виводити результати роботи кожної функції у зрозумілому для користувача форматі.

Рекомендується використовувати списки суміжності для представлення графа, оскільки це забезпечує ефективність для більшості алгоритмів обходу.

5. Порядок виконання

0. Підготовка:

- Створіть новий проект або файл для лабораторної роботи.
- Визначте, як будете представляти графи (клас `Graph`, клас `Node`, `Edge` тощо). Рекомендується використовувати `Map<Integer, List<Integer>>` або `List<List<Integer>>` для списків суміжності.

2. Реалізація базових функцій:

- Напишіть функцію для додавання ребер до графа (окремо для орієнтованих та неорієнтованих).

- Реалізуйте базовий DFS-обхід, який приймає граф, початкову вершину та масив `visited`.

3. Реалізація Компонент Зв’язності:
    - Напишіть функцію `findConnectedComponents(Graph graph)`.
    - Використайте в ній свій DFS-обхід, як описано в теоретичному матеріалі.
    - Протестуйте на неорієнтованому графі з кількома ізольованими компонентами.
4. Реалізація Мостів:
    - Напишіть функцію `findBridges(Graph graph)`.
    - Реалізуйте логіку з `discovery_time` та `low_link_value` за допомогою модифікованого DFS.
    - Протестуйте на неорієнтованому графі з очевидними мостами та без них.
5. Реалізація Точок Шарніру:
    - Напишіть функцію `findArticulationPoints(Graph graph)`.
    - Модифікуйте DFS з `discovery_time` та `low_link_value`, додавши обробку кореня та умову для не-кореневих вершин.
    - Протестуйте на неорієнтованому графі з різними конфігураціями точок шарніру.
6. Реалізація Топологічного Сортування (DFS-based):
    - Напишіть функцію `topologicalSortDFS(Graph graph)`.
    - Реалізуйте логіку DFS, яка додає вершини до стека після відвідування всіх нащадків.
    - Протестуйте на простому DAG.
7. Реалізація Топологічного Сортування (Алгоритм Кана):
    - Напишіть функцію `topologicalSortKahn(Graph graph)`.

- Реалізуйте логіку з обчисленням `in_degree` та використанням черги.
- Протестуйте на тому ж DAG.

8. Реалізація Пошуку Циклів:
   - Напишіть функцію `hasCycleUndirected(Graph graph)` для неорієнтованих графів.
   - Напишіть функцію `hasCycleDirected(Graph graph)` для орієнтованих графів з трьома станами відвідування.
   - Протестуйте на графах з циклами та без них.
9. Оформлення: Забезпечте зрозумілий вивід результатів для кожної функції.
10. Результат Після виконання всіх завдань програми повинні виводити:

– Для компонент зв'язності: Список компонентів, де кожна компонента — це список вершин (наприклад:

`[ [0, 1, 2], [3, 4] ]`).

– Для мостів: Список пар вершин, що утворюють мости (наприклад:

`[ (1, 3), (4, 5) ]`).

– Для точок шарніру: Список вершин, що є точками шарніру (наприклад: `[ 2, 4 ]`).

– Для топологічного сортування: Лінійний порядок вершин (наприклад: `[ 0, 1, 3, 2, 4 ]`). Якщо граф містить цикл, має бути виведено відповідне повідомлення.

– Для пошуку циклів: Логічне значення `true` або `false` та повідомлення, чи містить граф цикл.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:
   - Реалізовано структуру даних для представлення графа за допомогою списків суміжності.

- Надано окремі функції для кожного з алгоритмів: знаходження компонент зв’язності, мостів, точок шарніру, топологічного сортування (двома методами) та виявлення циклів (для орієнтованих та неорієнтованих графів).
- Код має бути добре структурованим, читабельним та містити необхідні коментарі.

2. Продемонструвати роботу додатку:
    - Продемонструвати роботу кожного алгоритму на прикладах графів, наданих викладачем, та на 1-2 власних тестових графах.
    - Показати коректність виводу результатів для кожного випадку.
    - Для топологічного сортування та пошуку циклів продемонструвати поведінку алгоритму як для графів без циклів, так і для графів з циклами.
3. Бути готовим відповісти на питання:
    - Пояснити вибір структур даних для реалізації графа.
    - Обґрунтувати часову та просторову складність кожного реалізованого алгоритму.
    - Детально описати, як працює виявлення мостів/точок шарніру з використанням `discovery_time` та `low_link_value`.
    - Пояснити, як алгоритми топологічного сортування (Кана та DFS-based) виявляють цикли.
    - Навести приклади реальних застосувань кожного з вивчених алгоритмів.

Тести

4. Яка структура даних зазвичай використовується для реалізації BFS?
   А) Стек

   Б) Черга

   В) Зв’язаний список

   Г) Хеш-таблиця

5. Що таке міст у неорієнтованому графі?

   А) Вершина, видалення якої роз’єднує граф.

   Б) Ребро, видалення якого збільшує кількість компонентів зв’язності.

   В) Ребро, що з’єднує дві відвідані вершини.

   Г) Найкоротший шлях між двома вершинами.

6. Яка умова є обов’язковою для графа, щоб його можна було топологічно відсортувати?

   А) Він має бути неорієнтованим.

   Б) Він має бути зваженим.

   В) Він має бути орієнтованим та ациклічним.

   Г) Він має містити щонайменше один цикл.

7. Який параметр вершини є ключовим для алгоритму Кана для топологічного сортування?

   А) Вихідний ступінь

   Б) Вхідний ступінь

   В) Вага ребра

   Г) Час відкриття

8. У пошуку циклів в орієнтованих графах за допомогою DFS, який стан вершини вказує на виявлення циклу, якщо ми зустрічаємо ребро, що веде до цієї вершини?

   А) `UNVISITED`

   Б) `VISITING`

   В) `VISITED`

   Г) `DISCOVERED`

Відповіді: 1. - Б), 2. - Б) 3. - В) 4. - Б) 5. - Б)

Тести підвищеної складності

1. Чому у неорієнтованих графах при пошуку циклів необхідно відстежувати батьківську вершину, тоді як в орієнтованих графах цього не вимагається для виявлення зворотних ребер?

   А) У неорієнтованих графах відсутня концепція “зворотного ребра” без батька.

   Б) В орієнтованих графах ребра мають напрямок, тому ребро до батька не існує.

   В) Щоб уникнути помилкового визначення ребра “дитина-батько” як циклу в неорієнтованих графах.

   Г) У неорієнтованих графах всі вершини є коренями DFS-дерева.

2. Якщо під час DFS-обходу неорієнтованого графа для ребра (U, V) виконується умова `low_link_value[V] == discovery_time[U]`, що це означає?

   А) Ребро (U, V) є мостом.

   Б) Вершина U є точкою шарніру.

   В) Вершина V є коренем DFS-дерева.

   Г) Існує шлях з піддерева V назад до U або її предка, проходячи через U.

3. В алгоритмі Кана, якщо після завершення обробки черги кількість оброблених вершин (`count`) менша за загальну кількість вершин у графі, що це означає?

4. А) Граф є незв’язним.

5. Б) Граф є лінійним шляхом.

6. В) Граф містить цикл.

7. Г) Виникла помилка вхідних даних.

8. Чому топологічне сортування на основі DFS додає вершини до стека лише після відвідування всіх її нащадків, а потім перевертає стек?

   А) Щоб забезпечити BFS-подібний порядок.

Б) Щоб дотримуватися принципу LIFO для стека.

В) Щоб гарантувати, що залежності (вершини, до яких ведуть ребра) будуть оброблені раніше.

Г) Для ефективнішого використання пам’яті.

9. У якій з наведених ситуацій використання алгоритму знаходження точок шарніру є найбільш доцільним?

   А) Оптимізація маршрутів у GPS-системах.

   Б) Виявлення критичних вузлів у комп’ютерній мережі, видалення яких може спричинити збій.

   В) Створення рекомендаційних систем для електронної комерції.

   Г) Моделювання поширення інформації у соціальних мережах.

10. Яка мінімальна кількість дочірніх піддерев повинна бути у кореневої вершини DFS-дерева, щоб вона вважалася точкою шарніру?

    А) 0

    Б) 1

    В) 2

    Г) Будь-яка кількість, якщо вона є коренем.

11. Яке твердження вірно щодо `low_link_value[U]` після його ініціалізації (`low_link_value[U] = discovery_time[U]`) та подальших оновлень у DFS для мостів/точок шарніру?

    А) Воно може тільки збільшуватися.

    Б) Воно може тільки зменшуватися.

    В) Воно завжди дорівнює `discovery_time[U]`.

    Г) Воно може як збільшуватися, так і зменшуватися залежно від сусідів.

12. Який з наступних графів не може бути топологічно відсортований?

А) Граф, що складається з однієї вершини.

Б) Граф, що складається з двох вершин, з’єднаних одним ребром.

В) Орієнтований граф, що містить ребро `A -> B`, `B -> C`, `C -> A`.

Г) Неорієнтований зв’язний граф.

13. Чому алгоритм знаходження компонент зв’язності повинен перевіряти всі вершини графа (від 0 до V-1) і запускати DFS/BFS з кожної невідвіданої вершини?

А) Щоб забезпечити правильний порядок обробки.

Б) Щоб знайти найкоротші шляхи між усіма вершинами.

В) Щоб коректно виявити всі ізольовані компоненти графа.

Г) Це необхідно лише для щільних графів.

14. У контексті пошуку циклів в орієнтованих графах, чи може ребро (U, V) вказувати на цикл, якщо `state[V]` дорівнює `VISITED`?

А) Так, завжди.

Б) Ні, ніколи.

В) Так, якщо `V` також є батьківською вершиною `U`.

Г) Це поперечне ребро або ребро в дереві DFS, що не вказує на цикл у поточному шляху.

Відповіді:

1 В). Щоб уникнути помилкового визначення ребра “дитина-батько” як циклу в неорієнтованих графах.

2 Г). Існує шлях з піддерева V назад до U або її предка, проходячи через U.

3 В). Граф містить цикл.

4 В). Щоб гарантувати, що залежності (вершини, до яких ведуть ребра) будуть оброблені раніше.

5 Б). Виявлення критичних вузлів у комп’ютерній мережі, видалення яких може спричинити збій.

6 В). 2.

7 Б). Воно може тільки зменшуватися.

8 В). Орієнтований граф, що містить ребро `A -> B`, `B -> C`, `C -> A`.

9 В). Щоб коректно виявити всі ізольовані компоненти графа.

10 Г). Це поперечне ребро або ребро в дереві DFS, що не вказує на цикл у поточному шляху.

Завдання для самостійної роботи

1. Ітеративний DFS: Перепишіть функцію `DFS_Visit` для обходу графа в глибину, використовуючи явний стек замість рекурсії. Порівняйте ваші враження від рекурсивної та ітеративної реалізацій.
2. BFS для найкоротшого шляху: Модифікуйте алгоритм BFS так, щоб він знаходив найкоротший шлях (за кількістю ребер) від заданої початкової вершини до всіх інших досяжних вершин у незваженому графі. Зберігайте батьківські вершини для відновлення шляху.
3. Сильно зв’язні компоненти: Дослідіть алгоритм Косараджу або Тар’яна для знаходження сильно зв’язних компонентів в орієнтованому графі. Спробуйте реалізувати один з них.
4. Зважений топологічний сорт: Припустіть, що вершини графа мають “пріоритети”. Модифікуйте алгоритм Кана або DFS-based сортування так, щоб вершини з вищим пріоритетом (при рівності вхідного ступеня) з’являлись раніше у топологічному порядку.
5. Ейлерові шляхи/цикли: Дослідіть критерії існування Ейлерового шляху або циклу в графі. Спробуйте реалізувати алгоритм для знаходження такого шляху/циклу, якщо він існує.
6. Граф залежностей: Візьміть реальний приклад залежностей (наприклад, між модулями програмного забезпечення, або етапами будівельного проекту) та змоделюйте його як орієнтований граф. Застосуйте топологічне сортування, щоб знайти можливий порядок виконання.
7. Двочасткові графи: Дослідіть, що таке двочастковий граф, та як BFS може бути використаний для перевірки, чи є граф двочастковим. Реалізуйте цей алгоритм.

8. Матриця суміжності для DFS/BFS: Модифікуйте свої реалізації DFS та BFS для роботи з графами, представленими за допомогою матриці суміжності. Порівняйте їхню продуктивність (часову складність) з версіями на списках суміжності.

9. Кількість циклів: Для орієнтованого графа, спробуйте модифікувати алгоритм пошуку циклів, щоб він не просто виявляв наявність циклу, а й рахував кількість унікальних циклів або виводив їх.

10. Проблема вершинного покриття: Дослідіть, що таке вершинне покриття, і як DFS може бути використаний як частина евристичного алгоритму для пошуку невеликого вершинного покриття. (Зверніть увагу: точне вершинне покриття є NP-складною задачею, зосередьтеся на евристиках).

Теми лабораторних робіт

1. Лабораторна робота №1: Реалізація основних обходів графа

    - Завдання: Реалізувати представлення графа за допомогою списків суміжності. Розробити функції для Пошуку в Ширину (BFS) та Пошуку в Глибину (DFS) для неорієнтованих графів. Продемонструвати порядок відвідування вершин для різних початкових точок.

2. Лабораторна робота №2: Виявлення компонент зв’язності

    - Завдання: На основі реалізованого DFS або BFS, розробити алгоритм для знаходження та виведення всіх компонентів зв’язності у заданому неорієнтованому графі. Протестувати на графах з однією та кількома компонентами.

3. Лабораторна робота №3: Знаходження мостів та точок шарніру

    - Завдання: Реалізувати алгоритми для виявлення всіх мостів та точок шарніру у неорієнтованому графі, використовуючи модифікований DFS (`discovery_time`, `low_link_value`). Продемонструвати їхню роботу на прикладах, що містять такі елементи.

4. Лабораторна робота №4: Топологічне сортування орієнтованих графів

- Завдання: Реалізувати два алгоритми топологічного сортування для орієнтованих ациклічних графів (DAG): алгоритм Кана та алгоритм на основі DFS. Продемонструвати їхню роботу та вивести отриманий порядок. Перевірити поведінку алгоритмів на графах, що містять цикли.

5. Лабораторна робота №5: Виявлення циклів у графах
    - Завдання: Розробити окремі функції для ефективного виявлення наявності циклів у неорієнтованих графах (використовуючи батьківську вершину) та в орієнтованих графах (використовуючи три стани відвідування). Протестувати на різних типах графів.
6. Лабораторна робота №6: Найкоротший шлях у незважених графах (BFS)
    - Завдання: Модифікувати алгоритм BFS для знаходження найкоротшого шляху (за кількістю ребер) від заданої початкової вершини до всіх інших досяжних вершин у незваженому графі. Зберігати інформацію про батьківські вершини для відновлення самого шляху.
7. Лабораторна робота №7: Алгоритм Дейкстри для найкоротших шляхів
    - Завдання: Реалізувати алгоритм Дейкстри для знаходження найкоротших шляхів від однієї початкової вершини до всіх інших вершин у зваженому орієнтованому графі з невід'ємними вагами ребер. Використати чергу з пріоритетом для оптимізації.
8. Лабораторна робота №8: Побудова мінімального остовного дерева (MST)
    - Завдання: Реалізувати один з алгоритмів побудови мінімального остовного дерева (наприклад, алгоритм Прима або алгоритм Крускала) для зв'язного неорієнтованого зваженого графа. Вивести вагу MST та список ребер, що його утворюють.

9. Лабораторна робота №9: Знаходження сильно зв’язних компонент (SCC)

    - Завдання: Дослідити та реалізувати алгоритм для знаходження сильно зв’язних компонент в орієнтованому графі (наприклад, алгоритм Косараджу або алгоритм Тар’яна). Продемонструвати виділення SCC на прикладах.

10. Лабораторна робота №10: Перевірка графа на двочастковість

    - Завдання: Розробити алгоритм, який перевіряє, чи є заданий неорієнтований граф двочастковим (біпартитним). Використати BFS або DFS для розфарбовування вершин у два кольори.

11. Лабораторна робота №11: Пошук Ейлерових шляхів/циклів

    - Завдання: Реалізувати алгоритм, який перевіряє, чи існує в заданому графі Ейлерів шлях або Ейлерів цикл, і якщо так, то виводить його. Розглянути як орієнтовані, так і неорієнтовані графи.

12. Лабораторна робота №12: Моделювання реальної задачі за допомогою графів

    - Завдання: Обрати просту реальну задачу (наприклад, планування завдань з залежностями, аналіз соціальної мережі, маршрутизація в невеликій транспортній мережі). Змоделювати її як граф і застосувати один або кілька вивчених алгоритмів для її розв’язання. Описати модель графа та обґрунтувати вибір алгоритму.

Сильно зв’язні компоненти (SCC): алгоритми Косараджу, Тар’яна.

У попередніх розділах ми вже згадували про компоненти зв’язності у неорієнтованих графах — це максимальні підграфи, де будь-яка пара вершин взаємно досяжна. Однак, для орієнтованих графів (де ребра мають напрямок) поняття “зв’язності” стає більш нюансованим.

Сильно зв’язна компонента (Strongly Connected Component, SCC) орієнтованого графа — це максимальний підграф, у якому для будь-яких

двох вершин U та V існує шлях як від U до V, так і від V до U. Це означає, що всередині SCC ви можете дістатися з будь-якої вершини до будь-якої іншої вершини, рухаючись лише вздовж напрямків ребер. Кожна вершина орієнтованого графа належить рівно до однієї сильно зв'язної компоненти.

Ключові відмінності від інших типів зв'язності:

- Зв'язні компоненти (неорієнтовані графи): Вимагають лише, щоб шлях існував в будь-якому напрямку (ігноруючи орієнтацію ребер).
- Слабо зв'язні компоненти (орієнтовані графи): Це компоненти зв'язності, отримані, якщо ігнорувати напрямки ребер у орієнтованому графі. Сильно зв'язні компоненти завжди є підмножиною слабо зв'язних компонент.

Поняття SCC є фундаментальним для розуміння загальної структури орієнтованого графа. Якщо ми "сколапсуємо" кожну SCC в одну "супервершину", а потім збережемо всі ребра між цими супервершинами, ми отримаємо новий граф, який називається графом конденсації (або SCC-графом). Важливою властивістю графа конденсації є те, що він завжди є орієнтованим ациклічним графом (DAG). Це означає, що в ньому не може бути циклів, і для нього можна виконати топологічне сортування. Ця властивість робить SCC потужним інструментом для спрощення аналізу складних орієнтованих графів.

1.1. Алгоритм Косараджу

Алгоритм Косараджу — це елегантний і відносно простий для розуміння алгоритм для знаходження сильно зв'язних компонент. Він базується на двох обходах графа в глибину (DFS).

Алгоритм складається з трьох основних кроків:

1. Перший DFS-обхід на оригінальному графі (G): Виконати DFS на початковому орієнтованому графі G. Під час цього обходу фіксується "час завершення" (або порядок, у якому вершини завершують свої рекурсивні виклики DFS). Вершини, які завершують обхід пізніше, зазвичай знаходяться у "вищих" SCC у графі конденсації.
2. Побудова транспонованого графа (G^T): Створити новий граф G^T, який є "транспонованим" або "оберненим" до G. Це означає, що для кожного ребра (U, V) у G ми створюємо ребро (V, U) у G^T. Тобто, всі напрямки ребер змінюються на протилежні.

3. Другий DFS-обхід на транспонованому графі (G^T): Виконати DFS на транспонованому графі G^T. Вершини для початку обходів обираються у порядку зменшення часу завершення, отриманого на кроці 1. Кожен DFS-обхід, запущений з невідвіданої вершини в G^T під час цього кроку, виявить усі вершини однієї сильно зв'язної компоненти.

Ідея полягає в тому, що якщо дві вершини U і V знаходяться в одній SCC, то існує шлях $U \rightsquigarrow V$ і шлях $V \rightsquigarrow U$. У транспонованому графі G^T також буде шлях $U \rightsquigarrow V$ і $V \rightsquigarrow U$. Порядок обходу на основі часу завершення гарантує, що ми завжди починаємо другий DFS-обхід з вершини, яка належить до "джерельної" SCC у графі конденсації (тобто, SCC, з якої немає вихідних ребер до інших SCC у графі конденсації). Оскільки всі ребра в G^T обернені, DFS-обхід з такої "джерельної" SCC в G^T буде здатний досягти всі вершини цієї SCC, але не зможе "втекти" до інших SCC (які в G^T виглядали б як "стоки").

Алгоритм Косараджу (покроково):

1. Ініціалізація:
    - Створити масив **`visited`** (булеві значення) для відстеження відвіданих вершин, ініціалізувати всі як **`false`**.
    - Створити порожній стек **`order_stack`** для зберігання вершин у порядку їх завершення першого DFS.
    - Ініціалізувати глобальний лічильник часу **`time = 0`** (хоча для простого порядку завершення він може бути не потрібен, якщо ми просто додаємо вершини до стека).
2. Перший прохід DFS на G:
    - Для кожної вершини i від 0 до $V - 1$:
        - Якщо **`visited[i]`** є **`false`**:
            - Викликати **`DFS_FIRST_PASS(G, i, visited, order_stack)`**.
    - Цей прохід заповнить **`order_stack`** вершинами у порядку їх завершення DFS (вершина, яка завершила обхід пізніше, буде вище в стеку).
3. Побудова транспонованого графа G^T:

- Створити новий граф `G_transpose` з тією ж кількістю вершин.
- Для кожної вершини U у G:
    - Для кожної сусідньої вершини V з U (тобто, для кожного ребра (U, V) у `G`):
        - Додати ребро (V, U) до `G_transpose`.

4. Ініціалізація для другого проходу:
    - Скинути масив `visited` (знову всі елементи `false`).
    - Створити порожній список `SCCs` для зберігання знайдених сильно зв'язних компонент.
5. Другий прохід DFS на G^T:
    - Поки `order_stack` не порожній:
        - Вилучити вершину U з вершини `order_stack`.
        - Якщо `visited[U]` є `false`:
            - Це означає, що ми знайшли нову SCC.
            - Створити порожній список `current_scc`.
            - Викликати `DFS_SECOND_PASS(G_transpose, U, visited, current_scc)`.
            - Додати `current_scc` до списку `SCCs`.
6. Результат: `SCCs` містить усі сильно зв'язні компоненти графа G.

Псевдокод алгоритму Косараджу

Структури даних, що використовуються:

– `Graph G`: Вхідний орієнтований граф.

– `AdjacencyList G.Adj`: Список суміжності для графа G.

– `Graph G_transpose`: Транспонований граф.

– `AdjacencyList G_transpose.Adj`: Список суміжності для G_transpose.

- `visited`: Булевий масив розміром `V`, ініціалізований `false`.
- `order_stack`: Стек для зберігання вершин у порядку завершення першого DFS.
- `SCCs`: Список списків, що містить усі знайдені SCC.
- `current_scc`: Тимчасовий список для збору вершин поточної SCC.

```
// Допоміжні операції:

// Операція: СТВОРИТИ_ПОРОЖНІЙ_ГРАФ(КІЛЬКІСТЬ_ВЕРШИН)
// Призначення: Створює новий граф з заданою кількістю вершин і без р
ебер.
// Повертає: Новий об'єкт Graph.
// Деталізація: Ініціалізує список суміжності (чи матрицю) відповідно
го розміру.

// Операція: ДОДАТИ_РЕБРО(Граф, ДЖЕРЕЛО, ПРИЗНАЧЕННЯ)
// Призначення: Додає орієнтоване ребро від ДЖЕРЕЛА до ПРИЗНАЧЕННЯ у
вказаний Граф.
// Деталізація: Для списку суміжності - додає ПРИЗНАЧЕННЯ до списку G
.Adj[ДЖЕРЕЛО].

// Операція: ДОДАТИ_В_СТЕК(Стек, Елемент)
// Призначення: Додає Елемент на вершину Стеку.
// Деталізація: Стандартна операція стеку (push).

// Операція: ВИДАЛИТИ_З_СТЕКУ(Стек)
// Призначення: Вилучає та повертає елемент з вершини Стеку.
// Повертає: Елемент з вершини Стеку.
// Деталізація: Стандартна операція стеку (pop).

// Операція: СТЕК_НЕ_ПОРОЖНІЙ(Стек)
// Призначення: Перевіряє, чи Стек містить елементи.
// Повертає: true, якщо Стек не порожній, false інакше.

// Операція: ПОЗНАЧИТИ_ВІДВІДАНУ(Масив_visited, Вершина, Значення)
// Призначення: Встановлює стан відвіданості Вершини у Масиві_visited
.
// Деталізація: Масив_visited[Вершина] = Значення.

// Операція: КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)
// Призначення: Повертає загальну кількість вершин у Графі.
```

```
// Операція: ДОДАТИ_В_СПИСОК(Список, Елемент)
// Призначення: Додає Елемент до кінця Списку.

// Операція: ОЧИСТИТИ_МАСИВ(Масив, Значення)
// Призначення: Встановлює всі елементи Масиву на задане Значення.

// Операція: ОТРИМАТИ_СУСІДІВ(Граф, Вершина)
// Призначення: Повертає список вершин, суміжних із заданою Вершиною.
// Деталізація: Для списків суміжності - G.Adj[Вершина].

АЛГОРИТМ KOSARAJU_SCC(Graph G):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН
  3. ОЧИСТИТИ_МАСИВ(visited, false) // Всі вершини спочатку не відвід ані.
  4. СТВОРИТИ ПОРОЖНІЙ Stack order_stack // Стек для порядку завершен ня першого DFS.

  5. // Крок 1: Перший DFS-обхід для заповнення order_stack
  6. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. ЯКЩО visited[i] == false:
          i. ВИКЛИКАТИ DFS_FIRST_PASS(G, i, visited, order_stack)

  7. // Крок 2: Побудова транспонованого графа G^T
  8. G_transpose = СТВОРИТИ_ПОРОЖНІЙ_ГРАФ(КІЛЬКІСТЬ_ВЕРШИН)
  9. ДЛЯ КОЖНОЇ ВЕРШИНИ U ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З G.Adj[U]): // Для кожно го ребра (U, V) в оригінальному графі
          i. ДОДАТИ_РЕБРО(G_transpose, V, U) // Додати обернене ребро (V, U) в транспонований граф

  10. // Крок 3: Другий DFS-обхід на G^T
  11. ОЧИСТИТИ_МАСИВ(visited, false) // Скинути стан відвіданості для нового обходу.
  12. СТВОРИТИ ПОРОЖНІЙ СПИСОК SCCs // Список для зберігання знайдени х SCC.

  13. ПОКИ СТЕК_НЕ_ПОРОЖНІЙ(order_stack):
      a. U = ВИДАЛИТИ_З_СТЕКУ(order_stack) // Вилучити вершину з верш ини стеку (порядок зменшення часу завершення).
      b. ЯКЩО visited[U] == false:
          i. СТВОРИТИ ПОРОЖНІЙ СПИСОК current_scc // Створити новий с писок для поточної SCC.
          ii. ВИКЛИКАТИ DFS_SECOND_PASS(G_transpose, U, visited, curr ent_scc) // Запустити DFS на транспонованому графі.
```

```
          iii. ДОДАТИ_В_СПИСОК(SCCs, current_scc) // Додати знайдену SCC до загального списку.

  14. ПОВЕРНУТИ SCCs

ФУНКЦІЯ DFS_FIRST_PASS(Graph G, Вершина U, Масив visited, Stack order_stack):
  1. ПОЗНАЧИТИ_ВІДВІДАНУ(visited, U, true) // Позначаємо U як відвідану.

  2. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З G.Adj[U]): // Проходимо по всіх сусідах U.
      a. ЯКЩО visited[V] == false:
          i. ВИКЛИКАТИ DFS_FIRST_PASS(G, V, visited, order_stack) // Рекурсивно викликаємо для V.

  3. ДОДАТИ_В_СТЕК(order_stack, U) // Додаємо U до стеку після завершення обходу всіх її нащадків.

ФУНКЦІЯ DFS_SECOND_PASS(Graph G_transpose, Вершина U, Масив visited, Список current_scc):
  1. ПОЗНАЧИТИ_ВІДВІДАНУ(visited, U, true) // Позначаємо U як відвідану.
  2. ДОДАТИ_В_СПИСОК(current_scc, U) // Додаємо U до поточної SCC.

  3. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З G_transpose.Adj[U]): // Проходимо по всіх сусідах U в транспонованому графі.
      a. ЯКЩО visited[V] == false:
          i. ВИКЛИКАТИ DFS_SECOND_PASS(G_transpose, V, visited, current_scc) // Рекурсивно викликаємо для V.
```

Операції та їх пояснення:

- **СТВОРИТИ_ПОРОЖНІЙ_ГРАФ(КІЛЬКІСТЬ_ВЕРШИН)**: Ця допоміжна операція ініціалізує новий об’єкт графа з заданою кількістю вершин, але без ребер. Наприклад, якщо ми використовуємо списки суміжності, це створить порожній список для кожної вершини.
- **ДОДАТИ_РЕБРО(Граф, ДЖЕРЕЛО, ПРИЗНАЧЕННЯ)**: Ця операція додає орієнтоване ребро від вершини **ДЖЕРЕЛО** до вершини **ПРИЗНАЧЕННЯ** у вказаний **Граф**. Якщо граф представлений списками суміжності, це просто додасть **ПРИЗНАЧЕННЯ** до списку, асоційованого з **ДЖЕРЕЛО**.

- `ДОДАТИ_В_СТЕК(Стек, Елемент)` / `ВИДАЛИТИ_З_СТЕКУ(Стек)` / `СТЕК_НЕ_ПОРОЖНІЙ(Стек)`: Це стандартні операції зі стеком (push, pop, isEmpty), які використовуються для управління `order_stack`.
- `ПОЗНАЧИТИ_ВІДВІДАНУ(Масив_visited, Вершина, Значення)`: Ця операція встановлює булеве значення (true/false) для вказаної `Вершини` в масиві `visited`. Це ключовий механізм для відстеження, які вершини вже були оброблені під час DFS, запобігаючи нескінченним циклам та повторній роботі.
- `ОЧИСТИТИ_МАСИВ(Масив, Значення)`: Допоміжна операція, яка встановлює кожен елемент заданого `Масиву` на вказане `Значення`. Використовується для скидання масиву `visited` між двома проходами DFS.
- `КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)`: Повертає загальну кількість вершин у графі.
- `ОТРИМАТИ_СУСІДІВ(Граф, Вершина)`: Повертає список усіх вершин, до яких існують орієнтовані ребра від заданої `Вершини` у вказаному `Графі`.
- `order_stack`: У першому проході `DFS_FIRST_PASS`, вершини додаються до цього стеку після того, як рекурсивний обхід з них повністю завершився. Це означає, що вершини, які “глибше” розташовані в дереві DFS і завершують обхід раніше, будуть знаходитись нижче в стеку, а вершини, які є “вищими” і завершують обхід пізніше, будуть на вершині стеку. Такий порядок є ключовим для другого проходу.
- `G_transpose`: Це транспонований граф, де всі ребра оригінального графа G змінили свій напрямок. Це дозволяє нам “слідувати” за зворотними шляхами в оригінальному графі, що є важливим для ізоляції SCC.
- `ПОКИ СТЕК_НЕ_ПОРОЖНІЙ(order_stack)`: Цей цикл у третьому кроці алгоритму є основним контролером. Він гарантує, що ми починаємо новий DFS-обхід у транспонованому графі з вершин, які мають найбільший “час завершення” з першого проходу. Це стратегія, яка дозволяє нам “захоплювати” цілі SCC, не переходячи в інші.

– `DFS_FIRST_PASS(...)`: Ця функція виконує перший DFS-обхід. Вона рекурсивно досліджує всі досяжні вершини, позначаючи їх як відвідані. Після того, як всі її нащадки були відвідані, і рекурсивні виклики для них повернулися, поточна вершина додається до `order_stack`.

– `DFS_SECOND_PASS(...)`: Ця функція виконує другий DFS-обхід на транспонованому графі. Вона також рекурсивно досліджує всі досяжні вершини, але вже в `G_transpose`. Усі вершини, які відвідані під час одного такого виклику, належать до однієї SCC. Вони додаються до `current_scc`.

Часова складність алгоритму Косараджу становить $O(V + E)$, де V — кількість вершин, а E — кількість ребер. Це пояснюється тим, що ми виконуємо два повних DFS-обходи (кожен $O(V + E)$) та один прохід для побудови транспонованого графа (також $O(V + E)$).

Просторова складність також $O(V + E)$ для зберігання графів та стеку/масивів.

1.2. Алгоритм Тар'яна

Алгоритм Тар'яна — це більш складний, але дуже елегантний і ефективний алгоритм для знаходження сильно зв'язних компонент. На відміну від алгоритму Косараджу, він використовує лише один DFS-обхід і не вимагає побудови транспонованого графа.

Алгоритм Тар'яна також базується на DFS і використовує два ключові значення для кожної вершини, подібні до тих, що ми бачили при знаходженні мостів та точок шарніру:

1. `discovery_time[U]` (час відкриття, `disc[U]`): Час (або порядковий номер), коли вершина U була вперше відвідана під час DFS.
2. `low_link_value[U]` (найнижче досяжне значення, `low[U]`): Мінімальний `discovery_time` серед усіх вершин W, які досяжні з U (включно з U) через ребра в дереві DFS, а також через одне зворотне ребро до будь-якого предка U (або самої U).

Крім цих двох значень, алгоритм Тар'яна використовує стек (не стек рекурсивних викликів, а допоміжний стек), який зберігає вершини, що знаходяться в поточному шляху DFS і які потенційно можуть належати до однієї SCC. Коли DFS повертається з обходу піддерева, алгоритм перевіряє

умову: якщо `low_link_value[U] == discovery_time[U]`, це означає, що вершина U є "коренем" (або найстарішою вершиною) однієї з SCC. Усі вершини, що знаходяться на стеку вище U (включно з U), і які були відвідані після U, належать до цієї ж SCC.

Алгоритм Тар'яна (покроково):

1. Ініціалізація:
    - Створити масив `discovery_time` та `low_link_value` для всіх вершин, ініціалізувати як нескінченність.
    - Створити булевий масив `on_stack` для відстеження, чи перебуває вершина у допоміжному стеку, ініціалізувати як `false`.
    - Створити порожній допоміжний стек `scc_stack`.
    - Створити порожній список `SCCs` для зберігання знайдених сильно зв'язних компонент.
    - Ініціалізувати глобальний лічильник часу `time = 0`.
2. Основний DFS-обхід:
    - Для кожної вершини i від 0 до $V - 1$:
        - Якщо `discovery_time[i]` є нескінченністю (тобто, вершина ще не була відвідана):
            - Викликати `DFS_TARJAN(G, i, time, discovery_time, low_link_value, on_stack, scc_stack, SCCs)`.
3. Визначення SCCs:
    - Вся логіка визначення SCCs інтегрована в `DFS_TARJAN` функцію.

Псевдокод алгоритму Тар'яна

Структури даних, що використовуються:

– `Graph G`: Вхідний орієнтований граф.

– `AdjacencyList G.Adj`: Список суміжності для графа G.

- `time`: Глобальний лічильник часу.
- `discovery_time`: Масив розміром `V`, що зберігає час відкриття кожної вершини.
- `low_link_value`: Масив розміром `V`, що зберігає найнижче досяжне значення для кожної вершини.
- `on_stack`: Булевий масив розміром `V`, `on_stack[U]` є `true`, якщо *U* перебуває у `scc_stack`.
- `scc_stack`: Допоміжний стек, що зберігає вершини, які наразі перебувають у поточному шляху DFS і ще не були призначені до SCC.
- `SCCs`: Список списків, що містить усі знайдені SCC.

```
// Допоміжні операції: (крім вже визначених у Косараджу)
// Операція: МІНІМУМ(Число1, Число2)
// Призначення: Повертає менше з двох чисел.

// Операція: ПОЗНАЧИТИ_НА_СТЕКУ(Масив_on_stack, Вершина, Значення)
// Призначення: Встановлює стан перебування Вершини у допоміжному сте
ку.
// Деталізація: Масив_on_stack[Вершина] = Значення.

// Операція: СТЕК_НЕ_ПОРОЖНІЙ(Стек)
// Призначення: Перевіряє, чи Стек містить елементи.
// Повертає: true, якщо Стек не порожній, false інакше.

// Операція: ОЧИСТИТИ_МАСИВ_ЗНАЧЕННЯ(Масив, Значення)
// Призначення: Встановлює всі елементи Масиву на задане Значення. На
приклад, нескінченність.

АЛГОРИТМ TARJAN_SCC(Graph G):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ МАСИВ discovery_time РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  3. СТВОРИТИ МАСИВ low_link_value РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  4. СТВОРИТИ БУЛЕВИЙ МАСИВ on_stack РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  5. СТВОРИТИ ПОРОЖНІЙ Stack scc_stack // Допоміжний стек
  6. СТВОРИТИ ПОРОЖНІЙ СПИСОК SCCs // Список для зберігання знайдених
SCC
  7. time = 0 // Глобальний лічильник часу

  8. ОЧИСТИТИ_МАСИВ_ЗНАЧЕННЯ(discovery_time, -1) // -1 або "нескінчен
ність" для невідвіданих
```

```
  9. ОЧИСТИТИ_МАСИВ(on_stack, false)

  10. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. ЯКЩО discovery_time[i] == -1: // Якщо вершина i ще не була в
ідвідана (її час відкриття ще не встановлений)
          i. ВИКЛИКАТИ DFS_TARJAN(G, i, time, discovery_time, low_lin
k_value, on_stack, scc_stack, SCCs)

  11. ПОВЕРНУТИ SCCs

ФУНКЦІЯ DFS_TARJAN(Graph G, Вершина U, time, discovery_time, low_link
_value, on_stack, scc_stack, SCCs):
  1. discovery_time[U] = time // Встановити час відкриття для U
  2. low_link_value[U] = time // Ініціалізувати low_link_value для U
її власним часом відкриття
  3. time = time + 1 // Збільшити глобальний лічильник часу
  4. ДОДАТИ_В_СТЕК(scc_stack, U) // Додати U до допоміжного стеку
  5. ПОЗНАЧИТИ_НА_СТЕКУ(on_stack, U, true) // Позначити U як таку, що
перебуває на стеку

  6. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З U (З G.Adj[U]): // Проходимо по
всіх сусідах U
      a. ЯКЩО discovery_time[V] == -1: // Якщо V ще не відвідана (ще
не має часу відкриття)
          i. ВИКЛИКАТИ DFS_TARJAN(G, V, time, discovery_time, low_lin
k_value, on_stack, scc_stack, SCCs) // Рекурсивно викликаємо для V
          ii. low_link_value[U] = МІНІМУМ(low_link_value[U], low_link
_value[V]) // Оновити low_link_value U, враховуючи найнижче досяжне з
начення з V
      b. ІНАКШЕ ЯКЩО on_stack[V] == true: // Якщо V вже відвідана і п
еребуває на стеку (зворотне ребро до предка в поточному шляху DFS)
          i. low_link_value[U] = МІНІМУМ(low_link_value[U], discovery
_time[V]) // Оновити low_link_value U з часом відкриття V

  7. // Якщо U є "коренем" SCC
  8. ЯКЩО low_link_value[U] == discovery_time[U]:
      a. СТВОРИТИ ПОРОЖНІЙ СПИСОК current_scc
      b. ПОКИ true: // Вилучаємо вершини зі стеку до U включно
          i. W = ВИДАЛИТИ_З_СТЕКУ(scc_stack)
          ii. ПОЗНАЧИТИ_НА_СТЕКУ(on_stack, W, false) // Позначити W я
к таку, що більше не перебуває на стеку
          iii. ДОДАТИ_В_СПИСОК(current_scc, W)
          iv. ЯКЩО W == U:
              1. ПЕРЕРВАТИ ЦИКЛ
```

```
    с. ДОДАТИ_В_СПИСОК(SCCs, current_scc) // Додати знайдену SCC до загального списку
```

Операції та їх пояснення:

- **`time`**: Глобальний лічильник, який використовується для присвоєння унікальних **`discovery_time`** кожній вершині при її першому відвідуванні.
- **`discovery_time[U]`**: Це **`час відкриття`** вершини U. Він встановлюється один раз при першому входженні в рекурсивний виклик для U.
- **`low_link_value[U]`**: Це **`найнижче досяжне значення`** для вершини U. Воно ініціалізується **`discovery_time[U]`**. Потім воно оновлюється (**`МІНІМУМ`**) з **`low_link_value`** дочірніх елементів (якщо вони досягають більш "ранніх" вершин) або з **`discovery_time`** вершин, до яких ведуть зворотні ребра і які все ще перебувають на **`scc_stack`**.
- **`scc_stack`**: Це допоміжний стек, який містить вершини в порядку їхнього відкриття під час поточного DFS-обходу. Він зберігає лише ті вершини, які ще не були призначені до жодної SCC. Коли SCC виявлена, всі її вершини вилучаються з цього стека.
- **`on_stack[U]`**: Булевий масив, який швидко вказує, чи перебуває вершина U на **`scc_stack`**. Це важливо для відрізнення зворотних ребер (що ведуть до предків у поточному шляху DFS) від поперечних ребер (що ведуть до вже повністю відвіданих вершин з іншої гілки DFS).
- **`ЯКЩО discovery_time[V] == -1`**: Ця умова перевіряє, чи сусідня вершина V ще не була відвідана. Якщо так, DFS рекурсивно занурюється у V.
- **`low_link_value[U] = МІНІМУМ(low_link_value[U], low_link_value[V])`**: Після рекурсивного виклику для V, ми оновлюємо **`low_link_value[U]`**. Це означає, що U може досягти будь-якої вершини, яку може досягти V (включно з її **`low_link_value`**), через ребро (U, V).

- **ЯКЩО `on_stack[V] == true`**: Ця умова є ключовою для виявлення зворотних ребер. Якщо сусід V вже був відвіданий (**`discovery_time[V]`** не -1) І він все ще перебуває на **`scc_stack`** (**`on_stack[V] == true`**), це означає, що V є предком U у поточному шляху DFS. Ребро (U, V) є зворотним ребром, і ми оновлюємо **`low_link_value[U]`** з **`discovery_time[V]`**, оскільки U може "повернутися" до V через це ребро.
- **ЯКЩО `low_link_value[U] == discovery_time[U]`**: Це основна умова для виявлення SCC. Якщо **`low_link_value`** вершини U дорівнює її **`discovery_time`**, це означає, що U є "коренем" (або найстарішою вершиною) SCC, і всі вершини, що були додані до **`scc_stack`** після U (включно з U), належать до цієї ж SCC. Жодна вершина з піддерева U не може досягти предків U через зворотні ребра, крім як через саму U.
- Вилучення зі **`scc_stack`**: Коли SCC виявлена (умова **`low_link_value[U] == discovery_time[U]`** істинна), усі вершини, починаючи з вершини стека і до U включно, вилучаються зі **`scc_stack`** і додаються до нової **`current_scc`**. Вони також позначаються як **`on_stack[W] = false`**.

Часова складність алгоритму Тар'яна також становить $O(V + E)$, оскільки виконується лише один повний DFS-обхід, і кожна вершина та ребро обробляються константу разів. Просторова складність також $O(V + E)$ для зберігання графа та додаткових масивів/стеку.

4. Область застосування

Сильно зв'язні компоненти є фундаментальним інструментом у багатьох галузях інформатики та не тільки. Їхня цінність полягає у здатності "розбити" складний орієнтований граф на простіші, ациклічні структури, що значно спрощує аналіз.

Основні області застосування включають:

1. Аналіз мереж та залежностей: SCC допомагають ідентифікувати циклічні залежності у складних системах.
2. Розробка програмного забезпечення: Виявлення взаємних залежностей між модулями, класами або функціями.

3. Бази даних: Оптимізація запитів та виявлення циклічних залежностей у схемах даних (наприклад, між таблицями).
4. Аналіз веб-графіків та соціальних мереж: Виявлення груп користувачів, які інтенсивно взаємодіють між собою, або “авторитетних” спільнот.
5. Біоінформатика: Аналіз мереж взаємодії білків або генних регуляторних мереж, де цикли можуть вказувати на певні біологічні процеси.
6. Системи контролю версій: Виявлення циклічних залежностей між репозиторіями або модулями.
7. Компілятори: Аналіз потоку управління в програмах для оптимізації коду, виявлення недосяжного коду або циклів виконання.
8. Перевірка властивостей систем: У моделюванні систем та перевірці моделей, SCC використовуються для виявлення досяжних станів та циклів у моделі.

5. Приклади застосування

Розглянемо декілька конкретних прикладів, які ілюструють практичну цінність SCC:

- Виявлення взаємних блокувань (Deadlock Detection) в операційних системах: У багатозадачних системах процеси можуть вимагати доступу до ресурсів, які зайняті іншими процесами. Це можна змоделювати як орієнтований граф ресурсів і процесів. Якщо у цьому графі є цикл, це свідчить про взаємне блокування. Хоча простий пошук циклів може виявити наявність блокування, знаходження SCC дозволяє ідентифікувати всі процеси та ресурси, які беруть участь у циклічному блокуванні, що є критичним для його розв’язання.
- Аналіз циклічних залежностей у програмних системах: При розробці великих програмних систем, часто виникають небажані циклічні залежності між модулями. Наприклад, модуль А залежить від модуля Б, а модуль Б залежить від модуля А. Такі цикли ускладнюють тестування, розгортання та підтримку коду. Побудувавши граф залежностей між модулями та знайшовши його SCC, розробники можуть точно визначити всі групи модулів, які мають циклічні

залежності. Це дозволяє рефакторити код для їхнього усунення або ізоляції.

- Рейтинг веб-сторінок або соціальних мереж: Хоча більш складні алгоритми, такі як PageRank, використовуються для ранжування, розуміння SCC може бути частиною аналізу. Групи веб-сторінок або користувачів у соціальній мережі, які сильно пов'язані між собою (тобто утворюють SCC), часто представляють спільноти або кластери, де інформація інтенсивно циркулює всередині, але може бути менш доступною ззовні. Це допомагає ідентифікувати "ехо-камери" або впливові спільноти.
- Моделювання біологічних процесів: У біологічних мережах (наприклад, мережах взаємодії білків або генетичних регуляторних мережах), орієнтовані графи використовуються для моделювання взаємодій. Знаходження SCC у таких мережах може вказувати на стійкі стани або цикли зворотного зв'язку, які є фундаментальними для клітинних процесів або механізмів розвитку захворювань.

Переваги та Недоліки

Переваги знаходження SCCs (загальні):

- Спрощення аналізу: Перетворення складного орієнтованого графа на граф конденсації (який є DAG) значно спрощує подальший аналіз. Багато алгоритмів, що працюють з DAG, є простішими та ефективнішими.
- Виявлення циклів: SCCs надають повний набір циклів у графі. Якщо граф має хоча б одну SCC розміром більше 1 (або 2, якщо дозволити петлі), він містить цикли.
- Структурний інсайт: Дозволяє ідентифікувати "жорсткі" взаємопов'язані кластери, які поводяться як єдині блоки в глобальній структурі графа.

Переваги та недоліки алгоритму Косараджу:

- Переваги:
    - Простота розуміння: Концептуально досить прямолінійний, складається з двох послідовних DFS-обходів та побудови

транспонованого графа. Це робить його добрим для початкового вивчення.

- Відносна простота реалізації: Оскільки він використовує лише стандартні DFS-обходи та операцію транспонування, реалізація може бути менш схильною до помилок порівняно з більш складними однопрохідними алгоритмами.

– Недоліки:

- Потрібна побудова транспонованого графа: Це вимагає додаткової пам'яті ($O(V + E)$) та додаткового часу для створення G^T. Хоча асимптотична складність не змінюється, на практиці це може бути менш ефективно для дуже великих графів, де пам'ять або час на транспонування є обмеженням.
- Два повних DFS-обходи: Потрібні два окремих обходи графа, що може бути повільніше на практиці, ніж однопрохідні алгоритми, через кеш-промахи та інші фактори.

Переваги та недоліки алгоритму Тар'яна:

– Переваги:

- Однопрохідний DFS: Алгоритм виконує лише один повний DFS-обхід, що робить його дуже ефективним з точки зору постійних факторів часу та часто кращим для практичної реалізації.
- Не вимагає транспонованого графа: Економить пам'ять та час, оскільки немає необхідності будувати G^T.
- Продуктивність: Часто є найшвидшим алгоритмом на практиці завдяки його однопрохідній природі та ефективному використанню стеку.

– Недоліки:

- Складність розуміння та реалізації: Логіка `low_link_value` та умови для виявлення SCCs є складнішими та вимагають глибокого розуміння роботи DFS та його взаємодії зі стеком. Це може призвести до більшої кількості помилок під час початкової реалізації.

- Використання додаткового стека: Потребує додаткового допоміжного стека, який може займати до $O(V)$ пам'яті.

Контрольні запитання

1. Визначення SCC: Чим сильно зв'язана компонента (SCC) відрізняється від звичайної зв'язної компоненти у неорієнтованому графі?
2. Граф конденсації: Що таке граф конденсації, і яка його ключова властивість, що робить його корисним для аналізу?
3. Косараджу - Кроки: Опишіть три основні кроки алгоритму Косараджу. Чому потрібен транспонований граф?
4. Косараджу - Перший DFS: Яку інформацію збирає перший DFS-обхід в алгоритмі Косараджу, і як ця інформація використовується у другому проході?
5. Тар'яна - Ключові значення: Які два ключові значення (`discovery_time` та `low_link_value`) використовуються в алгоритмі Тар'яна, і як `low_link_value` оновлюється?
6. Тар'яна - Стек: Для чого використовується допоміжний стек в алгоритмі Тар'яна? Коли вершини додаються та вилучаються з цього стека?
7. Тар'яна - Умова SCC: Яка умова (`discovery_time` та `low_link_value`) вказує на те, що вершина є "коренем" SCC в алгоритмі Тар'яна?
8. Переваги/Недоліки: Назвіть по одній головній перевазі та недоліку алгоритму Косараджу та алгоритму Тар'яна.
9. Застосування SCC: Наведіть два приклади реальних задач, де знаходження сильно зв'язних компонент є важливим.
10. Порівняння обходів: Як DFS використовується в обох алгоритмах SCC, і чим його роль відрізняється у Косараджу та Тар'яна?

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота №X: Виявлення Сильно Зв’язних Компонент в Орієнтованих Графах

2. Мета

Метою даної лабораторної роботи є здобуття глибокого розуміння та практичних навичок у реалізації алгоритмів для знаходження сильно зв’язних компонент (SCC) в орієнтованих графах. Ви будете реалізовувати та порівнювати два ключові алгоритми: Косараджу та Тар’яна.

3. Завдання

a) Реалізація представлення орієнтованого графа: Створити клас або структуру для представлення орієнтованого графа за допомогою списків суміжності.

b) Реалізація базового DFS: Розробити гнучку функцію для виконання обходу графа в глибину, яка може бути адаптована для різних цілей (наприклад, збирання часу завершення, використання `discovery_time`/`low_link_value`).

c) Реалізація алгоритму Косараджу:

   a. Реалізувати функцію для побудови транспонованого графа.

   b. Реалізувати алгоритм Косараджу для знаходження SCC, включаючи два проходи DFS.

d) Реалізація алгоритму Тар’яна:

   - Реалізувати алгоритм Тар’яна для знаходження SCC, включаючи логіку з `discovery_time`, `low_link_value` та допоміжним стеком.

4. Тестування: Протестувати обидва алгоритми на різних орієнтованих графах:

   - Граф без циклів (DAG).
   - Граф з одним великим циклом.
   - Граф з кількома окремими циклами, що формують різні SCC.
   - Граф з однією SCC, що охоплює більшість вершин.

5. Виведення результатів: Для кожного тестового графа чітко вивести знайдені SCC.
6. Опис

Вам необхідно розробити програмний додаток на обраній мові програмування (наприклад, Python, C++, Java, JavaScript). Додаток повинен дозволяти:

- Створювати орієнтовані графи з заданою кількістю вершин та ребер.
- Додавати орієнтовані ребра до графа.
- Викликати функції `findSCC_Kosaraju(Graph G)` та `findSCC_Tarjan(Graph G)`.
- Виводити результати роботи кожної функції у зрозумілому для користувача форматі, чітко відокремлюючи знайдені SCC.
- Включити в код допоміжні функції, такі як `printSCCs(List<List<Integer>> sccs)` для гарного форматування виводу.

Використання списків суміжності є обов'язковим для представлення графа.

7. Порядок Виконання
8. Налаштування середовища: Виберіть мову програмування та налаштуйте робоче середовище.
9. Реалізація класу/структури Графа:
    - Створіть клас `DirectedGraph` з полем для списків суміжності (`AdjacencyList`).
    - Реалізуйте метод `add_edge(U, V)` для додавання орієнтованих ребер.
    - Реалізуйте метод `get_transpose()` або `build_transpose_graph()` для створення транспонованого графа.
10. Реалізація базового DFS-хелпера:
    - Напишіть гнучку рекурсивну функцію `DFS_Visit(Graph G, int U, boolean[] visited, ...)` яка може приймати

додаткові параметри для збору інформації або модифікації поведінки (наприклад, стек для Косараджу, `discovery_time`, `low_link_value` для Тар'яна).

11. Реалізація алгоритму Косараджу:
    - Створіть функцію `findSCC_Kosaraju(Graph G)`.
    - Імплементуйте три кроки алгоритму, як описано в теорії, використовуючи вашу функцію `DFS_Visit` (або її специфічні версії `DFS_FIRST_PASS`, `DFS_SECOND_PASS`).
12. Реалізація алгоритму Тар'яна:
    - Створіть функцію `findSCC_Tarjan(Graph G)`.
    - Імплементуйте логіку одного проходу DFS, використовуючи `discovery_time`, `low_link_value`, `scc_stack` та `on_stack`.
13. Тестування:
    - Створіть кілька екземплярів `DirectedGraph` для різних тестових сценаріїв (як описано в Завданнях, пункт 5).
    - Для кожного графа викликайте обидві функції `findSCC_Kosaraju` та `findSCC_Tarjan`.
    - Виведіть результати у зрозумілому форматі, наприклад:

      ```
      Граф X:
      SCCs за алгоритмом Косараджу: [[0, 1, 2], [3], [4, 5]]
      SCCs за алгоритмом Тар'яна:   [[2, 1, 0], [3], [5, 4]] // П
      орядок всередині SCC та порядок SCCs може відрізнятись, це
      нормально
      ```
14. Оформлення: Забезпечте зрозумілий інтерфейс для створення графів та виклику алгоритмів.
15. Результат

Після виконання всіх завдань програми повинні виводити:

– Для кожного тестового графа:
  - Список знайдених сильно зв'язних компонент за алгоритмом Косараджу.

- Список знайдених сильно зв’язних компонент за алгоритмом Тар’яна.
- Кожна SCC має бути представлена як окремий список вершин.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:
    - Реалізовано структуру даних для представлення орієнтованого графа за допомогою списків суміжності.
    - Надано окремі, повністю функціональні реалізації алгоритмів Косараджу та Тар’яна для знаходження SCC.
    - Код має бути добре структурованим, читабельним та містити необхідні коментарі, що пояснюють ключові частини реалізації (наприклад, призначення `discovery_time`, `low_link_value`, `on_stack`).
2. Продемонструвати роботу додатку:
    - Продемонструвати роботу обох алгоритмів на прикладах графів, наданих викладачем, та на 2-3 власних тестових графах, що ілюструють різні випадки (наприклад, граф без SCCs (DAG), граф з однією великою SCC, граф з декількома малими SCCs).
    - Показати коректність виводу результатів для кожного тестового випадку, вказуючи, які вершини належать до кожної SCC.
3. Бути готовим відповісти на питання:
    - Вибір представлення: Обґрунтувати вибір списків суміжності для представлення графа.
    - Складність: Детально пояснити часову та просторову складність кожного реалізованого алгоритму (Косараджу, Тар’яна).
    - Принципи роботи: Пояснити, чому алгоритм Косараджу потребує двох проходів DFS та транспонованого графа. Пояснити, як `discovery_time`, `low_link_value` та стек взаємодіють в алгоритмі Тар’яна для виявлення SCC.

- Порівняння: Порівняти обидва алгоритми з точки зору їхніх переваг та недоліків, а також їхньої практичної застосовності.
- Граф конденсації: Пояснити, що таке граф конденсації, та як він може бути побудований після знаходження SCC.
- Застосування: Навести 2-3 приклади реальних задач, де застосування знаходження сильно зв’язних компонент є критично важливим.

Тести

1. Що є визначенням Сильно Зв’язної Компоненти (SCC) в орієнтованому графі?

   А) Максимальний підграф, де будь-які дві вершини з’єднані шляхом (ігноруючи напрямок).

   Б) Максимальний підграф, де для будь-яких двох вершин U і V існує шлях від U до V.

   В) Максимальний підграф, де для будь-яких двох вершин U і V існує шлях як від U до V, так і від V до U.

   Г) Будь-яка група вершин, що утворює цикл.

2. Який алгоритм знаходження SCC вимагає побудови транспонованого графа?

   А) Алгоритм Тар’яна

   Б) Алгоритм Кана

   В) Алгоритм Косараджу

   Г) Алгоритм Дейкстри

3. В алгоритмі Косараджу, для чого використовується перший DFS-обхід?

   А) Для знаходження найкоротших шляхів до всіх вершин.

   Б) Для обчислення вхідного ступеня кожної вершини.

   В) Для визначення порядку завершення вершин у DFS-обході.

Г) Для виявлення циклів у графі.

4. Які два ключові значення використовуються в алгоритмі Тар'яна для ідентифікації SCC?

   А) Вхідний ступінь та Вихідний ступінь

   Б) Відстань та Попередник

   В) Час відкриття та Найнижче досяжне значення

   Г) Час початку та Час завершення

5. Граф конденсації орієнтованого графа завжди є:
   А) Неорієнтованим графом.
   Б) Зваженим графом.
   В) Орієнтованим ациклічним графом (DAG).
   Г) Повним графом.

Відповіді: 1 - В), 2 - В), 3 - В), 4 - В), 5 - В).

Тести підвищеної складності

1. Чому у другому проході DFS алгоритму Косараджу обхід виконується на транспонованому графі та у порядку зменшення часу завершення з першого проходу?

   А) Щоб гарантувати, що ми починаємо з вершин з найменшими ID.

   Б) Щоб дозволити DFS обходити лише в межах однієї SCC.

   В) Щоб уникнути циклів у транспонованому графі.

   Г) Щоб забезпечити BFS-подібний обхід.

2. Яке твердження вірно щодо `low_link_value[U]` в алгоритмі Тар'яна?

   А) Воно завжди дорівнює `discovery_time[U]`.

   Б) Воно є мінімальним `discovery_time` серед усіх вершин, досяжних з `U` у поточному піддереві DFS.

   В) Воно є максимальним `discovery_time` серед усіх вершин, досяжних з `U` через зворотне ребро.

Г) Воно є мінімальним `discovery_time` серед `U` та всіх вершин, досяжних з `U` через дерево DFS або одне зворотне ребро до предка, який перебуває на стеку.

3. В алгоритмі Тар'яна, коли вершина *U* вилучається з допоміжного стека і позначається як `on_stack = false`?

   А) Після того, як всі її сусіди були відвідані.

   Б) Коли вона додається до стека.

   В) Коли вона ідентифікується як "корінь" нової SCC, і всі її компоненти вилучаються зі стека.

   Г) Після завершення всього DFS-обходу графа.

4. Уявіть, що ви використовуєте SCCs для виявлення циклічних залежностей між модулями програмного забезпечення. Що ви робитимете після знаходження SCCs?

   А) Просто вилучите всі модулі, що входять до SCC.

   Б) Спробуєте реорганізувати модулі всередині SCC, щоб усунути циклічні залежності.

   В) Застосуєте алгоритм Дейкстри до SCC.

   Г) Запустите топологічне сортування безпосередньо на оригінальному графі.

5. Який з наведених алгоритмів є однопрохідним (вимагає лише одного повного DFS-обходу) для знаходження SCC?

   А) Алгоритм Косараджу.

   Б) Алгоритм Кана.

   В) Алгоритм Тар'яна.

   Г) Алгоритм Флойда-Воршелла.

6. Якщо в орієнтованому графі знайдена SCC складається лише з однієї вершини, що це означає?

   А) Ця вершина є мостом.

Б) Ця вершина є точкою шарніру.

В) Ця вершина не бере участі в жодному циклі.

Г) Ця вершина є ізольованою.

7. Чому порядок вилучення вершин зі стека у другому DFS-обході алгоритму Косараджу є критичним?

   А) Він дозволяє обходити граф від вершин з найменшим `discovery_time`.

   Б) Він гарантує, що ми завжди починаємо DFS у SCC, з якої не виходить ребер до інших SCC у графі конденсації.

   В) Він дозволяє виявити цикли.

   Г) Він забезпечує випадковий порядок обходу.

8. Яка з цих структур даних є ключовою для алгоритму Тар'яна для тимчасового зберігання вершин, що ще не були призначені до SCC?

   А) Черга з пріоритетом.

   Б) Масив `visited`.

   В) Допоміжний стек.

   Г) Хеш-таблиця `discovery_time`.

9. Яка часова складність алгоритмів Косараджу та Тар'яна для знаходження SCC, якщо граф представлений списками суміжності?

   А) $O(V^2)$

   Б) $O(E \log V)$

   В) $O(V + E)$

   Г) $O(V \cdot E)$

10. В чому полягає основна відмінність між `discovery_time` і `low_link_value` в алгоритмі Тар'яна, коли вони використовуються для ідентифікації SCC?

А) `discovery_time` - це час, коли вершина була додана до стека, `low_link_value` - коли вилучена.

Б) `discovery_time` - це час першого відвідування, `low_link_value` - найнижчий час відкриття, досяжний з цієї вершини через дерево або одне зворотне ребро.

В) `discovery_time` - це час завершення обходу піддерева, `low_link_value` - час початку обходу.

Г) `discovery_time` - це кількість ребер до вершини, `low_link_value` - кількість вершин у піддереві.

Відповіді:

1 Б). Щоб дозволити DFS обходити лише в межах однієї SCC.

2 Г). Воно є мінімальним `discovery_time` серед `U` та всіх вершин, досяжних з `U` через дерево DFS або одне зворотне ребро до предка, який перебуває на стеку.

3 В). Коли вона ідентифікується як “корінь” нової SCC, і всі її компоненти вилучаються зі стека.

4 Б). Спробуєте реорганізувати модулі всередині SCC, щоб усунути циклічні залежності.

5 В). Алгоритм Тар’яна.

6 В). Ця вершина не бере участі в жодному циклі.

7 Б). Він гарантує, що ми завжди починаємо DFS у SCC, з якої не виходить ребер до інших SCC у графі конденсації.

8 В). Допоміжний стек.

9 В). $O(V + E)$.

10 Б). `discovery_time` - це час першого відвідування, `low_link_value` - найнижчий час відкриття, досяжний з цієї вершини через дерево або одне зворотне ребро.

Завдання для самостійної роботи

1. Візуалізація SCC: Після реалізації алгоритмів Косараджу та Тар’яна, спробуйте візуалізувати знайдені SCC. Наприклад, виведіть кожну

SCC різним кольором або згрупуйте вершини, що належать до однієї SCC.

2. Побудова графа конденсації: На основі знайдених SCC, побудуйте граф конденсації. Виведіть його структуру (списки суміжності супервершин). Перевірте, чи є він завжди DAG.

3. Виявлення циклів через SCC: Використайте знаходження SCC як метод для виявлення наявності циклів в орієнтованому графі. Поясніть, як це працює.

4. Аналіз залежностей у коді: Оберіть невеликий проект на GitHub або частину власного коду. Створіть граф залежностей між функціями/класами (де ребро $A \to B$ означає, що A викликає/використовує B). Знайдіть SCC у цьому графі та проаналізуйте їх.

5. Алгоритм Флойда-Воршелла для досяжності: Дослідіть алгоритм Флойда-Воршелла, який може обчислювати транзитивне замикання графа (тобто, чи існує шлях між будь-якими двома парами вершин). Поясніть, як це пов’язано з SCC.

6. SCC для топологічного сортування графів з циклами: Поясніть, як знаходження SCC може бути використано для “топологічного сортування” графа, що містить цикли, шляхом сортування його графа конденсації.

7. Ітеративна реалізація Тар’яна: Спробуйте переписати алгоритм Тар’яна в ітеративному вигляді, використовуючи явний стек замість рекурсії.

8. Оптимізація пам’яті для транспонованого графа: Якщо граф дуже великий, побудова транспонованого графа може споживати багато пам’яті. Дослідіть методи, які дозволяють уникнути явного створення G^T для алгоритму Косараджу, якщо це можливо (наприклад, обхід графа, читаючи його ребра у зворотному порядку).

9. Алгоритм для 2-зв’язних компонент: Дослідіть, що таке 2-зв’язні компоненти (або блоки), та як вони пов’язані з точками шарніру та мостами. Спробуйте знайти алгоритм для їхнього виявлення.

10. Алгоритм SCC з випадковим порядком: Замість послідовного перебору вершин у першому DFS (для Косараджу) або головному циклі (для Тар'яна), спробуйте розпочати обходи у випадковому порядку. Як це впливає на продуктивність та коректність алгоритмів?

Теми лабораторних робіт

1. Лабораторна робота №1: Реалізація основних обходів графа
    - Завдання: Реалізуй представлення графа за допомогою списків суміжності. Розроби функції для Пошуку в Ширину (BFS) та Пошуку в Глибину (DFS) для неорієнтованих графів. Продемонструй порядок відвідування вершин для різних початкових точок.
2. Лабораторна робота №2: Аналіз зв'язності графа
    - Завдання: Реалізуй алгоритми для знаходження компонентів зв'язності у неорієнтованому графі та слабо зв'язних компонент в орієнтованому графі. Проаналізуй, як ці компоненти впливають на загальну структуру графа.
3. Лабораторна робота №3: Виявлення критичних елементів мережі
    - Завдання: Реалізуй алгоритми для виявлення всіх мостів та точок шарніру у неорієнтованому графі, використовуючи модифікований DFS (`discovery_time`, `low_link_value`). Продемонструй їхню роботу на прикладах, що містять такі елементи.
4. Лабораторна робота №4: Порівняння алгоритмів топологічного сортування
    - Завдання: Реалізуй та порівняй два алгоритми топологічного сортування для орієнтованих ациклічних графів (DAG): алгоритм Кана та алгоритм на основі DFS. Проведи аналіз їхньої продуктивності на різних типах DAG (наприклад, лінійний, розгалужений).
5. Лабораторна робота №5: Розширений пошук циклів
    - Завдання: Розроби окремі функції для ефективного виявлення наявності циклів у неорієнтованих та орієнтованих графах. Для

орієнтованих графів, окрім простого виявлення, спробуй вивести один зі знайдених циклів.

6. Лабораторна робота №6: Найкоротший шлях у незважених графах (BFS)
    - Завдання: Модифікуй алгоритм BFS для знаходження найкоротшого шляху (за кількістю ребер) від заданої початкової вершини до всіх інших досяжних вершин у незваженому графі. Зберігай інформацію про батьківські вершини для відновлення самого шляху.
7. Лабораторна робота №7: Алгоритм Дейкстри для найкоротших шляхів
    - Завдання: Реалізуй алгоритм Дейкстри для знаходження найкоротших шляхів від однієї початкової вершини до всіх інших вершин у зваженому орієнтованому графі з невід'ємними вагами ребер. Використай чергу з пріоритетом для оптимізації.
8. Лабораторна робота №8: Алгоритми мінімального остовного дерева (Прим та Крускал)
    - Завдання: Реалізуй обидва алгоритми побудови мінімального остовного дерева (MST): алгоритм Прима та алгоритм Крускала. Виведи вагу MST та список ребер, що його утворюють.
9. Лабораторна робота №9: Знаходження сильно зв'язних компонент (Косараджу та Тар'яна)
    - Завдання: Реалізуй обидва алгоритми для знаходження сильно зв'язних компонент (SCC) в орієнтованому графі: алгоритм Косараджу та алгоритм Тар'яна. Продемонструй їхню роботу на складних прикладах.
10. Лабораторна робота №12: Моделювання та аналіз реальної мережі
    - Завдання: Обери реальну мережу (наприклад, невелику соціальну мережу, мережу доріг, мережу залежностей завдань у проєкті). Побудуй її граф-модель. Застосуй 2-3 вивчені алгоритми (наприклад, BFS для найкоротшого шляху, DFS для

компонент зв’язності, або SCC для виявлення кластерів) для аналізу цієї мережі та інтерпретуй отримані результати.

Ейлерові та Гамільтонові шляхи/цикли.

Теоретичні відомості

Поняття Ейлерового шляху виникло з історичної задачі про сім мостів Кенігсберга, яку ми вже розглядали. Ця задача поклала початок теорії графів.

Ейлерів шлях у графі (орієнтованому або неорієнтованому) — це шлях, який проходить через кожне ребро графа рівно один раз. Ейлерів цикл — це Ейлерів шлях, який починається і закінчується в одній і тій же вершині.

Важливо розуміти, що Ейлерів шлях/цикл стосується саме ребер: кожне ребро має бути використане точно один раз. Вершини можуть бути відвідані кілька разів. Граф, що містить Ейлерів цикл, називається Ейлеровим графом. Граф, що містить Ейлерів шлях, але не цикл, називається напівейлеровим графом.

Існують чіткі критерії для визначення, чи має граф Ейлерів шлях або цикл. Ці критерії залежать від властивостей вершин графа.

Для неорієнтованого графа:

- Граф має Ейлерів цикл, якщо і тільки якщо:
    1. Він є зв’язним (усі вершини належать до однієї компоненти зв’язності, інакше ребра ізольованих частин не можуть бути обійдені).
    2. Кожна вершина має парний ступінь (кількість ребер, що інцидентні вершині).
        - Пояснення: Якщо вершина має непарний ступінь, то ви не зможете “ввійти” в неї і “вийти” з неї, використовуючи кожне ребро рівно один раз, не залишаючись у ній (для проміжної вершини) або не починаючи/закінчуючи обхід у ній (для початкової/кінцевої вершини циклу).
- Граф має Ейлерів шлях (але не цикл), якщо і тільки якщо:
    1. Він є зв’язним.

2. Рівно дві вершини мають непарний ступінь. Ці дві вершини будуть початковою та кінцевою точками Ейлерового шляху.
    - Пояснення: Усі проміжні вершини на шляху повинні мати парний ступінь, оскільки ви “входите” в них по одному ребру і “виходите” по іншому. Якщо ж є дві вершини з непарним ступенем, одна з них може бути початком шляху, а інша — кінцем.

Для орієнтованого графа:

– Граф має Ейлерів цикл, якщо і тільки якщо:
    1. Він є сильно зв’язним (якщо ми ігноруємо напрямки ребер, базовий неорієнтований граф є зв’язним, і крім того, усі вершини належать до однієї SCC, або всі ненульові вершини належать до однієї SCC).
    2. Для кожної вершини *V*, її вхідний ступінь дорівнює її вихідному ступеню (`in_degree(V) == out_degree(V)`).
        - Пояснення: Коли ви входите в вершину, ви повинні також вийти з неї, використовуючи одне ребро, якщо ви не починаєте або не закінчуєте цикл у ній. Для циклу це правило діє для всіх вершин.
– Граф має Ейлерів шлях (але не цикл), якщо і тільки якщо:
    1. Базовий неорієнтований граф є зв’язним (якщо ігноруємо напрямки ребер).
    2. Існує одна вершина *S*, для якої `out_degree(S) = in_degree(S) + 1` (початкова вершина шляху).
    3. Існує одна вершина *T*, для якої `in_degree(T) = out_degree(T) + 1` (кінцева вершина шляху).
    4. Для всіх інших вершин *V* (`in_degree(V) == out_degree(V)`).
        - Пояснення: Початкова вершина матиме на одне вихідне ребро більше, кінцева — на одне вхідне більше.

Алгоритм для знаходження Ейлерового шляху/циклу (Флері)

Алгоритм Флері є одним з методів знаходження Ейлерового циклу, якщо він існує.

Почніть з довільної вершини (для циклу) або з вершини з непарним ступенем (для шляху). Послідовно переміщуйтеся по ребрах, додаючи їх до поточного шляху. Ключове правило: ніколи не обирайте ребро, видалення якого роз'єднає граф, якщо тільки це ребро не є єдиним, що залишилося від поточної вершини. Це дозволяє уникнути застрягання в ізольованій частині графа до того, як усі ребра будуть використані.

Алгоритм Флері (для неорієнтованих графів, спрощений):

1. Попередня перевірка: Виконати перевірку на існування Ейлерового шляху/циклу за критеріями, описаними вище. Якщо умови не виконуються, шлях/цикл не існує.
2. Визначення початкової вершини: Якщо існує Ейлерів цикл, оберіть довільну вершину. Якщо існує Ейлерів шлях (з двома непарними ступенями), оберіть одну з вершин з непарним ступенем як початкову.
3. Ініціалізація:
    - Створити порожній список `eulerian_path` для зберігання вершин шляху.
    - Створити "робочий" граф `current_graph`, який є копією вихідного графа. У ньому ми будемо видаляти ребра після їх використання.
    - Ініціалізувати поточну вершину `current_vertex` як початкову вершину.
    - Позначити всі ребра як невикористані.
4. Обхід:
    - Додати `current_vertex` до `eulerian_path`.
    - Доки у `current_graph` є невикористані ребра:
        - Знайти наступне ребро ($current_vertex, next_vertex$) для переходу. Правило вибору:

- Якщо є кілька невикористаних ребер з `current_vertex`: обрати те ребро, видалення якого не робить `current_graph` незв'язним (тобто, не є мостом у залишковому `current_graph`).
- Якщо всі ребра з `current_vertex` є мостами (у залишковому графі), крім одного, обрати це єдине ребро.
- Якщо існує лише одне невикористане ребро з `current_vertex`, обрати його.

▪ Якщо такого ребра не знайдено (наприклад, вершина isolated, але ребра залишились десь інде), це помилка або граф не Ейлерів.

▪ Видалити ребро $(current_vertex, next_vertex)$ з `current_graph`.

▪ Оновити `current_vertex = next_vertex`.

▪ Додати `current_vertex` до `eulerian_path`.

5. Завершення: Якщо після обходу всі ребра були використані, `eulerian_path` містить Ейлерів шлях/цикл.

Псевдокод Алгоритму Флері (спрощений)

Цей псевдокод фокусується на логіці обходу, припускаючи наявність допоміжних функцій для перевірки зв'язності та видалення ребер.

Структури даних, що використовуються:

– `Graph G`: Вхідний граф (неорієнтований або орієнтований).

– `AdjacencyList G.Adj`: Список суміжності графа.

– `degrees`: Масив ступенів вершин (для неорієнтованого), або `in_degree`/`out_degree` (для орієнтованого).

– `edge_used`: Булевий масив або мапа для відстеження використаних ребер.

– `eulerian_path`: Список для зберігання вершин Ейлерового шляху/циклу.

```
// Допоміжні функції:
// ПОРАХУВАТИ_СТУПЕНІ(Graph G): Обчислює ступені (in/out для орієнтов
аних) всіх вершин.
// ПЕРЕВІРИТИ_ЧИ_ЗВ'ЯЗНИЙ(Graph G): Використовує DFS/BFS для перевірк
и зв'язності.
// ПЕРЕВІРИТИ_ЧИ_МІСТ(Граф G, Ребро(U,V)): Перевіряє, чи є ребро (U,V
) мостом у Графі G. (Ця функція сама може використовувати DFS/BFS і п
ідрахунок компонент зв'язності)
// ВИДАЛИТИ_РЕБРО_З_ГРАФА(Граф G, Ребро(U,V)): Логічно видаляє ребро
з графа (наприклад, знімає позначку is_used або видаляє з списків сум
іжності).

АЛГОРИТМ FIND_EULERIAN_PATH(Graph G):
  1. // Попередня перевірка критеріїв існування
  2. ПОРАХУВАТИ_СТУПЕНІ(G) // Оновлює масиви degrees/in_degree/out_de
gree
  3. ЯКЩО ГРАФ_НЕ_ВІДПОВІДАЄ_КРИТЕРІЯМ_ЕЙЛЕРА(G): // Використання ран
іше описаних правил
      a. ПОВЕРНУТИ "Ейлерів шлях/цикл не існує"

  4. ЗНАЙТИ_ПОЧАТКОВУ_ВЕРШИНУ = визначити початкову вершину за критер
іями (довільна для циклу, непарна для шляху)
  5. СТВОРИТИ ПОРОЖНІЙ СПИСОК eulerian_path
  6. СТВОРИТИ КОПІЮ_ГРАФА G_TEMP = G // Для модифікації під час обход
у

  7. current_vertex = ЗНАЙТИ_ПОЧАТКОВУ_ВЕРШИНУ
  8. ДОДАТИ current_vertex В eulerian_path

  9. ПОКИ В G_TEMP Є НЕВИКОРИСТАНІ РЕБРА:
      a. ЗНАЙТИ_НАСТУПНЕ_РЕБРО = NULL
      b. ДЛЯ КОЖНОГО НЕВИКОРИСТАНОГО РЕБРА (current_vertex, V) З G_TE
MP:
          i. ЯКЩО (НЕ ПЕРЕВІРИТИ_ЧИ_МІСТ(G_TEMP, (current_vertex, V))
) АБО (КІЛЬКІСТЬ_НЕВИКОРИСТАНИХ_РЕБЕР_З_current_vertex(G_TEMP) == 1):
              1. ЗНАЙТИ_НАСТУПНЕ_РЕБРО = (current_vertex, V)
              2. ПЕРЕРВАТИ ЦИКЛ

      c. ЯКЩО ЗНАЙТИ_НАСТУПНЕ_РЕБРО == NULL:
          i. ПОВЕРНУТИ "Помилка: Неможливо знайти наступне ребро (гра
ф не Ейлерів)"

      d. next_vertex = ЗНАЙТИ_НАСТУПНЕ_РЕБРО.V // Кінцева вершина обр
аного ребра
      e. ВИДАЛИТИ_РЕБРО_З_ГРАФА(G_TEMP, ЗНАЙТИ_НАСТУПНЕ_РЕБРО) // Поз
```

```
начити ребро як використане/видалити
      f. current_vertex = next_vertex
      g. ДОДАТИ current_vertex В eulerian_path

  10. ПОВЕРНУТИ eulerian_path
```

Операції та їх пояснення (для алгоритму Флері):

- `ПОРАХУВАТИ_СТУПЕНІ(Graph G)`: Ця функція обчислює ступінь кожної вершини для неорієнтованих графів (або вхідний/вихідний ступінь для орієнтованих). Це потрібно для попередньої перевірки критеріїв Ейлера.
- `ГРАФ_НЕ_ВІДПОВІДАЄ_КРИТЕРІЯМ_ЕЙЛЕРА(G)`: Допоміжна функція, яка інкапсулює логіку перевірки умов існування Ейлерового шляху/циклу, описаних вище. Вона включає перевірку зв'язності (за допомогою DFS/BFS) та аналіз ступенів вершин.
- `ЗНАЙТИ_ПОЧАТКОВУ_ВЕРШИНУ`: Функція, яка визначає, з якої вершини розпочати обхід: довільна вершина для циклу, або одна з двох вершин з непарним ступенем для шляху.
- `СТВОРИТИ КОПІЮ_ГРАФА G_TEMP = G`: Алгоритм Флері вимагає модифікації графа (видалення ребер). Тому ми працюємо з копією, щоб не змінювати оригінальний граф.
- `ПОКИ В G_TEMP Є НЕВИКОРИСТАНІ РЕБРА`: Основний цикл алгоритму, який продовжується до тих пір, поки всі ребра не будуть використані.
- `ПЕРЕВІРИТИ_ЧИ_МІСТ(G_TEMP, (current_vertex, V))`: Це важлива допоміжна функція. Вона перевіряє, чи є ребро $(current_vertex, V)$ мостом у поточному стані графа `G_TEMP` (тобто, серед ребер, що ще не були використані). Це вимагає окремого DFS/BFS та підрахунку компонент зв'язності після тимчасового видалення ребра, що робить цей алгоритм не дуже ефективним.
- `КІЛЬКІСТЬ_НЕВИКОРИСТАНИХ_РЕБЕР_З_current_vertex(G_TEMP)`: Функція, що повертає кількість невикористаних ребер, що виходять з `current_vertex` у `G_TEMP`.
- Правило вибору ребра: Ключова евристика Флері. Вона гарантує, що ми не "застрягнемо" у частині графа, залишивши невикористані

ребра в іншій частині. Ми обираємо міст тільки в тому випадку, якщо це єдине ребро, що залишилося від поточної вершини.

- `ВИДАЛИТИ_РЕБРО_З_ГРАФА(G_ТЕМР, ЗНАЙТИ_НАСТУПНЕ_РЕБРО)`: Логічне видалення ребра означає, що воно більше не буде розглядатися.

Часова складність: Алгоритм Флері має часову складність $O(E \cdot (V + E))$ або $O(E^2)$ у випадку використання матриці суміжності для перевірки мостів. Це пов'язано з тим, що перевірка, чи є ребро мостом, сама по собі вимагає обходу графа ($O(V + E)$), і це повторюється для кожного ребра. Це робить його неефективним для великих графів.

Існують більш ефективні алгоритми, такі як алгоритм Ієрхольцера, який працює за $O(V + E)$ часу. Його ідея полягає у послідовному пошуку циклів, видаленні їх з графа, а потім з'єднанні цих циклів в один великий Ейлерів цикл.

1.2. Гамільтонові шляхи та цикли

На відміну від Ейлерових шляхів/циклів, які зосереджені на ребрах, Гамільтонові шляхи/цикли зосереджені на вершинах.

Гамільтонів шлях у графі (орієнтованому або неорієнтованому) — це шлях, який відвідує кожну вершину графа рівно один раз. Гамільтонів цикл — це Гамільтонів шлях, який починається і закінчується в одній і тій же вершині.

Гамільтонів шлях/цикл стосується саме вершин: кожна вершина має бути відвідана точно один раз. Ребра можуть бути використані не всі. Граф, що містить Гамільтонів цикл, називається Гамільтоновим графом.

На відміну від Ейлерових шляхів/циклів, для Гамільтонових шляхів/циклів немає простих і загальних критеріїв, які б дозволяли легко визначити їхнє існування. Це одна з причин, чому задачі, пов'язані з Гамільтоновими шляхами, набагато складніші.

Існують лише деякі достатні умови (тобто, якщо умова виконується, Гамільтонів шлях/цикл існує, але якщо вона не виконується, це не означає, що шляху/циклу немає), наприклад, теорема Дірака або теорема Оре, які стосуються мінімального ступеня вершин. Однак, ці теореми не є необхідними і не допомагають знайти шлях/цикл.

Задачі пошуку Гамільтонових шляхів/циклів належать до класу NP-повних задач. Це означає, що наразі невідомі поліноміальні алгоритми для їхнього

розв’язання. Найкращі відомі алгоритми мають експоненціальну часову складність у найгіршому випадку, що робить їх непридатними для великих графів.

Алгоритми для знаходження Гамільтонового шляху/циклу

Оскільки задача є NP-повною, алгоритми для її розв’язання, як правило, використовують методи пошуку з поверненнями або динамічного програмування (на основі бітових масок) для невеликих графів.

Алгоритм пошуку з поверненнями:

Цей алгоритм систематично перевіряє всі можливі шляхи, доки не знайде Гамільтонів шлях/цикл або не вичерпає всі варіанти.

Основна ідея:

1. Почніть з довільної вершини.
2. Спробуйте розширити поточний шлях, переходячи до невідвіданих сусідніх вершин.
3. Якщо шлях досягає всіх вершин, перевірте, чи це Гамільтонів шлях. Якщо шукається цикл, перевірте, чи існує ребро від останньої вершини до початкової.
4. Якщо поточний шлях не може бути розширений або не веде до рішення, поверніться (backtrack) до попередньої вершини і спробуйте іншу гілку.

Алгоритм (для Гамільтонового циклу, неорієнтований граф):

1. Ініціалізація:
    - Створити масив `path` розміром V, ініціалізувати всі елементи як `-1` (для зберігання вершин у поточному шляху).
    - Створити булевий масив `visited` розміром V, ініціалізувати всі як `false`.
    - Встановити `path[0] = start_vertex` (довільна початкова вершина).
    - Позначити `visited[start_vertex] = true`.
2. Викликати рекурсивну функцію `FIND_HAMILTONIAN_CYCLE(G, path, visited, 1)`.

Функція `FIND_HAMILTONIAN_CYCLE(Graph G, Масив path, Масив visited, int position)`:

3. Базовий випадок (успіх):
    - ЯКЩО `position == КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)`: // Якщо всі вершини відвідані
        - ЯКЩО ІСНУЄ_РЕБРО(G, path[position - 1], path[0]): // Перевірити, чи є ребро до початкової вершини для циклу
            - ВИВЕСТИ_ШЛЯХ(path)
            - ПОВЕРНУТИ true // Гамільтонів цикл знайдено
        - ІНАКШЕ:
            - ПОВЕРНУТИ false // Це шлях, але не цикл
4. Рекурсивний крок:
    - ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З path[position - 1] (З G.Adj[path[position - 1]]): // Перебираємо сусідів останньої вершини в шляху
        - ЯКЩО visited[V] == false: // Якщо V ще не відвідана
            - path[position] = V // Додати V до шляху
            - visited[V] = true // Позначити V як відвідану
            - ЯКЩО ВИКЛИКАТИ FIND_HAMILTONIAN_CYCLE(G, path, visited, position + 1) ПОВЕРТАЄ true: // Рекурсивно спробувати розширити шлях
                - ПОВЕРНУТИ true // Цикл знайдено
            - visited[V] = false // Відкат: зняти позначку відвіданості (для інших гілок пошуку)
            - path[position] = -1 // Відкат: вилучити V зі шляху
5. ПОВЕРНУТИ false // Жоден шлях з цієї позиції не привів до Гамільтонового циклу

Псевдокод Алгоритму пошуку гамільтонового циклу

Структури даних, що використовуються:

- `Graph G`: Вхідний граф.
- `AdjacencyList G.Adj`: Список суміжності графа.
- `path`: Масив розміром `V`, що зберігає поточний Гамільтонів шлях.
- `visited`: Булевий масив розміром `V`, що відстежує відвідані вершини в поточному шляху.

```
// Допоміжні функції:
// ІСНУЄ_РЕБРО(Граф, Вершина1, Вершина2): Перевіряє наявність ребра м
іж двома вершинами.
// ВИВЕСТИ_ШЛЯХ(Масив_path): Виводить знайдений Гамільтонів шлях.

АЛГОРИТМ FIND_HAMILTONIAN_CYCLE_MAIN(Graph G):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ МАСИВ path РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН
  3. СТВОРИТИ БУЛЕВИЙ МАСИВ visited РОЗМІРОМ З КІЛЬКІСТЮ_ВЕРШИН
  4. ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ path ЯК -1
  5. ІНІЦІАЛІЗУВАТИ ВСІ ЕЛЕМЕНТИ visited ЯК false

  6. // Можна почати з будь-якої вершини, наприклад, 0
  7. path[0] = 0 // Перша вершина у шляху
  8. visited[0] = true // Позначаємо її як відвідану

  9. ЯКЩО ВИКЛИКАТИ FIND_HAMILTONIAN_CYCLE_RECURSIVE(G, path, visited
, 1) ПОВЕРТАЄ true:
      a. ПОВЕРНУТИ "Гамільтонів цикл знайдено"
  10. ІНАКШЕ:
      a. ПОВЕРНУТИ "Гамільтонів цикл не знайдено"

ФУНКЦІЯ FIND_HAMILTONIAN_CYCLE_RECURSIVE(Graph G, Масив path, Масив v
isited, int position):
  1. // Базовий випадок: Шлях містить всі вершини
  2. ЯКЩО position == КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G):
      a. ЯКЩО ІСНУЄ_РЕБРО(G, path[position - 1], path[0]): // Перевір
яємо, чи є ребро від останньої до початкової вершини
          i. ВИВЕСТИ_ШЛЯХ(path) // Виводимо знайдений цикл
          ii. ПОВЕРНУТИ true
      b. ІНАКШЕ:
          i. ПОВЕРНУТИ false // Шлях знайдено, але це не цикл

  3. // Рекурсивний крок: Спробувати розширити шлях
  4. ДЛЯ КОЖНОЇ СУМІЖНОЇ ВЕРШИНИ V З path[position - 1] (З G.Adj[path
```

```
[position - 1]]):
      a. ЯКЩО visited[V] == false: // Якщо V ще не відвідана у поточн
ому шляху
          i. path[position] = V // Додаємо V до шляху
          ii. visited[V] = true // Позначаємо V як відвідану
          iii. ЯКЩО ВИКЛИКАТИ FIND_HAMILTONIAN_CYCLE_RECURSIVE(G, pat
h, visited, position + 1) ПОВЕРТАЄ true: // Рекурсивно намагаємось зн
айти решту шляху
              1. ПОВЕРНУТИ true
          iv. visited[V] = false // Відкат (backtracking): якщо цей ш
лях не привів до рішення, знімаємо позначку відвіданості
          v. path[position] = -1 // Відкат: видаляємо вершину зі шлях
у

  5. ПОВЕРНУТИ false // Жоден шлях, що починається з цієї позиції, не
привів до Гамільтонового циклу
```

Операції та їх пояснення (для Гамільтонового циклу, Backtracking):

- `path`: Масив, який зберігає послідовність вершин у поточному шляху, який ми будуємо. `path[k]` буде k-тою вершиною в шляху.
- `visited`: Булевий масив, який відстежує, які вершини вже були включені до поточного шляху. Це відрізняється від `visited` у DFS для обходів, оскільки тут ми повинні відвідати кожну вершину лише один раз.
- `position`: Параметр рекурсивної функції, що вказує на поточну позицію (індекс) у масиві `path`, яку ми намагаємося заповнити.
- `ІСНУЄ_РЕБРО(G, Вершина1, Вершина2)`: Допоміжна функція, яка перевіряє, чи існує пряме ребро між `Вершина1` та `Вершина2` у графі G. Це потрібно для перевірки з'єднання між останньою вершиною шляху та початковою для формування циклу.
- `ВИВЕСТИ_ШЛЯХ(Масив_path)`: Допоміжна функція для виведення знайденого Гамільтонового циклу.
- Базовий випадок (`position == КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)`): Це умова, яка перевіряє, чи ми успішно додали всі V вершин до шляху. Якщо так, ми маємо потенційний Гамільтонів шлях.
- Перевірка на цикл (`ІСНУЄ_РЕБРО(G, path[position - 1], path[0])`): Якщо ми досягли всіх вершин, ми перевіряємо, чи існує ребро від останньої вершини в шляху (`path[position - 1]`) до

початкової вершини (`path[0]`). Якщо так, ми знайшли Гамільтонів цикл.

- `visited[V] == false`: Це умова для розширення шляху. Ми можемо переходити лише до вершин, які ще не були відвідані у поточному шляху.
- `path[position] = V` / `visited[V] = true`: Додаємо вершину V до шляху та позначаємо її як відвідану.
- `visited[V] = false` / `path[position] = -1` (Відкат): Це ключовий крок пошуку з поверненнями. Якщо поточна гілка пошуку не призвела до рішення (рекурсивний виклик повернув `false`), ми "відкочуємо" зміни: знімаємо позначку відвіданості з V і видаляємо її зі шляху. Це дозволяє алгоритму спробувати інші варіанти.

Часова складність: У найгіршому випадку алгоритм пошуку з поверненнями перевіряє всі можливі шляхи. Його часова складність є експоненціальною, приблизно $O(V \cdot E)$ або $O(V \cdot V)$, що робить його придатним лише для дуже малих графів (до 20-25 вершин).

5. **Область застосування**

Ейлерові шляхи/цикли:

- Планування маршрутів та логістика: Наприклад, маршрут снігоприбиральної машини, яка повинна пройти по кожній вулиці міста рівно один раз; маршрут поштаря; інспекція доріг.
- Обслуговування мереж: Перевірка всіх кабелів у мережі без повторного проходження.
- Біологічні додатки: Збірка геномів (побудова контигів з фрагментів ДНК).
- Малювання однією лінією: Старі головоломки, де потрібно намалювати фігуру, не відриваючи олівця від паперу і не проводячи по одній лінії двічі.

Гамільтонові шляхи/цикли:

- Задача комівояжера (Travelling Salesperson Problem, TSP): Хоча TSP є складнішою задачею (зважений Гамільтонів цикл з пошуком мінімальної загальної ваги), знаходження Гамільтонового циклу є її

підзадачею. Це стосується планування найкоротших маршрутів, що відвідують кожне місто рівно один раз.

- Біоінформатика: Аналіз ДНК (пошук послідовностей ДНК, де кожен нуклеотид або кодон відвідано один раз).
- Проектування мікросхем: Маршрутизація доріжок на платі, де кожна точка має бути відвідана.
- Мережева безпека: Виявлення вразливостей, які дозволяють пройти по всіх вузлах мережі.
- Ігри та головоломки: Наприклад, “хід коня” на шахівниці, де кінь повинен відвідати кожну клітинку рівно один раз.

4. Приклади застосування

Приклад Ейлерового шляху:

Уявіть собі систему роботів, які повинні провести інспекцію всіх трубопроводів на нафтопереробному заводі. Кожен трубопровід представляє ребро, а місця з’єднань — вершини. Якщо роботи почнуть з певного з’єднання, пройдуть по кожному трубопроводу рівно один раз і повернуться до початкового з’єднання, це буде Ейлерів цикл. Якщо їм потрібно пройти по всіх трубопроводах, але вони можуть закінчити в іншому місці, це буде Ейлерів шлях. Це дозволяє мінімізувати час і енерговитрати, уникаючи повторного проходження одних і тих же ділянок.

Приклад Гамільтонового циклу:

Найвідоміший приклад — Задача комівояжера. Хоча вона часто включає ваги (відстані/вартості), її базовою ідеєю є Гамільтонів цикл. Уявіть, що кур’єр повинен доставити посилки у 10 різних точок міста, відвідавши кожну точку рівно один раз, а потім повернутися на склад. Кожна точка доставки — це вершина, а дороги між ними — ребра. Кур’єр шукає Гамільтонів цикл. Оскільки ця задача є складною, для її практичного вирішення (з мінімізацією відстані) використовуються складніші алгоритми та евристики.

Переваги та Недоліки

Ейлерові шляхи/цикли:

- Переваги:

- Існують прості та ефективні алгоритми для їхнього знаходження ($O(V + E)$).
- Легко перевірити умови існування, що дозволяє швидко визначити, чи задача має рішення.
- Пряме застосування в задачах обходу мереж без повторень.

– Недоліки:

- Застосовні лише до графів, що відповідають дуже специфічним критеріям ступенів вершин.
- Не можуть бути використані для задач, де важливо відвідати кожну вершину (а не кожне ребро) рівно один раз.

Гамільтонові шляхи/цикли:

– Переваги:

- Моделюють широкий клас задач, де кожна “точка” має бути відвідана точно один раз.
- Мають значні застосування в логістиці, біології, проєктуванні.

– Недоліки:

- Задача є NP-повною: Це означає, що немає відомих ефективних (поліноміальних) алгоритмів для її розв’язання. Для великих графів час виконання експоненціально зростає.
- Відсутність простих критеріїв існування: Немає легких умов, як для Ейлерових шляхів, щоб визначити, чи існує Гамільтонів шлях/цикл у графі.
- Вимагають експоненціального пошуку: Навіть для графів середнього розміру, пошук з поверненнями може бути занадто повільним. Для практичних застосувань часто використовуються евристики або апроксимаційні алгоритми.

Контрольні Запитання

1. Основна відмінність: Яка принципова відмінність між Ейлеровим шляхом/циклом та Гамільтоновим шляхом/циклом з точки зору того, що вони відвідують?

2. Критерії Ейлера (неорієнтований): Які умови повинні бути виконані для неорієнтованого графа, щоб він мав Ейлерів цикл? А Ейлерів шлях, але не цикл?
3. Критерії Ейлера (орієнтований): Які умови повинні бути виконані для орієнтованого графа, щоб він мав Ейлерів цикл? А Ейлерів шлях, але не цикл?
4. Алгоритм Флері: Яке ключове правило застосовується в алгоритмі Флері для вибору наступного ребра, щоб уникнути "застрягання"?
5. NP-повна задача: До якого класу складності належить задача пошуку Гамільтонового шляху/циклу? Що це означає для практичної реалізації?
6. Метод пошуку Гамільтона: Опишіть основну ідею алгоритму пошуку Гамільтонового циклу з використанням пошуку з поверненнями (backtracking).
7. Відкат у Гамільтоновому пошуку: Навіщо потрібна операція "відкат" (`visited[V] = false`, `path[position] = -1`) в алгоритмі пошуку Гамільтонового циклу з поверненнями?
8. Область застосування Ейлера: Наведіть один приклад реальної задачі, де Ейлерів шлях/цикл є корисним.
9. Область застосування Гамільтона: Наведіть один приклад реальної задачі, де Гамільтонів шлях/цикл (або його варіації) є корисним.
10. Порівняння складності: Чому Ейлерові шляхи/цикли можна знайти за поліноміальний час, тоді як Гамільтонові — ні (для великих графів)?

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота №X: Дослідження Ейлерових та Гамільтонових Шляхів/Циклів

2. Мета

Метою даної лабораторної роботи є набуття теоретичного розуміння та практичних навичок у реалізації алгоритмів для пошуку Ейлерових

шляхів/циклів, а також вивчення підходів до пошуку Гамільтонових шляхів/циклів у графах.

3. Завдання

a) Реалізація представлення графа: Створити клас або структуру для представлення неорієнтованого графа за допомогою списків суміжності.

b) Перевірка критеріїв Ейлера: Реалізувати функцію, яка перевіряє, чи заданий неорієнтований граф має Ейлерів цикл або Ейлерів шлях (але не цикл).

c) Пошук Ейлерового шляху/циклу: Реалізувати алгоритм Флері (або більш ефективний алгоритм Ієрхольцера, якщо вистачить часу) для знаходження та виведення Ейлерового шляху/циклу, якщо він існує.

d) Пошук Гамільтонового циклу: Реалізувати алгоритм пошуку Гамільтонового циклу за допомогою методу пошуку з поверненнями (backtracking).

e) Тестування: Протестувати реалізовані алгоритми на наданих прикладах графів та на власних тестових даних.

4. Опис

Вам необхідно розробити програмний додаток на обраній мові програмування (наприклад, Python, C++, Java, JavaScript). Додаток повинен дозволяти:

- Створювати неорієнтовані графи з заданою кількістю вершин та ребер.
- Додавати ребра до графа.
- Викликати функції для перевірки критеріїв Ейлера та пошуку Ейлерових шляхів/циклів.
- Викликати функцію для пошуку Гамільтонового циклу.
- Виводити результати роботи кожної функції у зрозумілому для користувача форматі.

Рекомендується використовувати списки суміжності для представлення графа.

5. Порядок Виконання
6. Підготовка:
    - Створіть новий проект або файл для лабораторної роботи.
    - Визначте, як будете представляти неорієнтовані графи (клас `Graph`, клас `Node`, `Edge` тощо).
7. Реалізація допоміжних функцій:
    - Функція для додавання ребер до неорієнтованого графа.
    - Функція для обчислення ступенів вершин (`calculate_degrees`).
    - Функція для перевірки зв'язності графа (наприклад, за допомогою BFS або DFS).
8. Реалізація перевірки критеріїв Ейлера:
    - Створіть функцію `checkEulerian(Graph G)` яка повертає тип Ейлерового графа (цикл, шлях, жоден) на основі ступенів вершин та зв'язності.
9. Реалізація пошуку Ейлерового шляху/циклу:
    - Створіть функцію `findEulerianPath(Graph G)`.
    - Імплементуйте алгоритм Флері (або Ієрхольцера). Пам'ятайте, що для Флері потрібна функція `is_bridge(Graph G, Edge e)` яка перевіряє, чи є ребро мостом.
    - Протестуйте на графах, що відповідають різним критеріям Ейлера (мають цикл, мають шлях, не мають нічого).
10. Реалізація пошуку Гамільтонового циклу (Backtracking):
    - Створіть функцію `findHamiltonianCycle(Graph G)` з рекурсивним допоміжним методом.
    - Реалізуйте логіку пошуку з поверненнями, використовуючи масиви `path` та `visited`.

- Протестуйте на невеликих графах (до 10-15 вершин), де відомо, чи існує Гамільтонів цикл.

11. Оформлення: Забезпечте зрозумілий вивід результатів для кожної функції.

12. Результат

Після виконання всіх завдань програми повинні виводити:

– Для Ейлерових шляхів/циклів:

  - Результат перевірки критеріїв (наприклад, “Граф має Ейлерів цикл”, “Граф має Ейлерів шлях (між X та Y)”, “Граф не має Ейлерового шляху/циклу”).
  - Якщо шлях/цикл існує, вивести послідовність вершин/ребер, що його утворюють.

– Для Гамільтонових циклів:

  - Повідомлення, чи знайдено Гамільтонів цикл.
  - Якщо знайдено, вивести послідовність вершин циклу.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:
   - Реалізовано структуру даних для представлення неорієнтованого графа за допомогою списків суміжності.
   - Надано функцію для перевірки критеріїв Ейлера.
   - Надано функцію для пошуку Ейлерового шляху/циклу.
   - Надано функцію для пошуку Гамільтонового циклу за допомогою backtracking.
   - Код має бути добре структурованим, читабельним та містити необхідні коментарі.
2. Продемонструвати роботу додатку:

- Продемонструвати роботу алгоритму пошуку Ейлерових шляхів/циклів на 2-3 прикладах графів, що ілюструють різні випадки (цикл, шлях, відсутність).
- Продемонструвати роботу алгоритму пошуку Гамільтонових циклів на 2-3 невеликих графах, де відомо про наявність/відсутність циклу.
- Показати коректність виводу результатів для кожного тестового випадку.

3. Бути готовим відповісти на питання:
    - Принципи Ейлера: Пояснити критерії існування Ейлерового шляху/циклу для неорієнтованих та орієнтованих графів.
    - Принципи Флері: Описати логіку алгоритму Флері, включаючи правило вибору ребра та необхідність перевірки на міст.
    - Суть Гамільтона: Дати визначення Гамільтонового шляху/циклу та пояснити, чим вони відрізняються від Ейлерових.
    - NP-повні задачі: Пояснити, що означає, що задача Гамільтонового циклу є NP-повною, і які наслідки це має для її практичного розв’язання.
    - Метод Backtracking: Детально пояснити, як працює метод пошуку з поверненнями в контексті пошуку Гамільтонового циклу, включаючи поняття “відкату”.
    - Складність: Обґрунтувати часову складність реалізованих алгоритмів.

Тести

1. Ейлерів шлях у графі відвідує кожне:

    А) Ребро рівно один раз.

    Б) Ребро щонайменше один раз.

    В) Вершину рівно один раз.

    Г) Вершину щонайменше один раз.

2. Яка умова є необхідною для існування Ейлерового циклу в неорієнтованому зв’язному графі?

   А) Всі вершини мають непарний ступінь.

   Б) Всі вершини мають парний ступінь.

   В) Рівно дві вершини мають непарний ступінь.

   Г) Граф є двочастковим.

3. Гамільтонів цикл у графі відвідує кожну:

   А) Вершину рівно один раз.

   Б) Вершину щонайменше один раз.

   В) Ребро рівно один раз.

   Г) Ребро щонайменше один раз.

4. Задача пошуку Гамільтонового циклу належить до класу:

   А) P

   Б) NP-повних

   В) NP-легких

   Г) L

5. В алгоритмі пошуку Гамільтонового циклу методом backtracking, якщо поточна гілка пошуку не привела до рішення, виконується операція:

   А) Розширення

   Б) Обрізання

   В) Відкат

   Г) Обхід в ширину

Відповіді:1 - А), 2 - Б), 3 - А), 4 - Б),5 - В).

Тести підвищеної складності

1. Чому алгоритм Флері для знаходження Ейлерового шляху може бути неефективним для великих графів?

   А) Через використання рекурсії.

   Б) Через необхідність побудови транспонованого графа.

   В) Через часту перевірку на наявність мостів.

   Г) Через експоненційну кількість можливих шляхів.

2. Яке твердження є правильним для орієнтованого графа, що має Ейлерів шлях (але не цикл)?

   А) Вхідний ступінь кожної вершини дорівнює її вихідному ступеню.

   Б) Існує одна вершина S з `out_degree(S) = in_degree(S) + 1` і одна вершина T з `in_degree(T) = out_degree(T) + 1`, а інші мають рівні ступені.

   В) Всі вершини мають парний ступінь.

   Г) Він обов’язково є сильно зв’язним.

3. Яка з наведених задач може бути безпосередньо змодельована як пошук Ейлерового шляху?

   А) Маршрут кур’єра, який має відвідати 10 поштових відділень.

   Б) Оптимальний маршрут робота-прибиральника, який має проїхати кожною ділянкою підлоги складу.

   В) Найкоротший маршрут з однієї точки в іншу в системі GPS.

   Г) Розклад занять, де кожен предмет має бути вивчений лише один раз.

4. Що є ключовою відмінністю між `visited` масивами в стандартному DFS (для обходу) та в алгоритмі пошуку Гамільтонового циклу backtracking?

   А) У DFS `visited` відстежує всі відвідані вершини, у Гамільтоновому пошуку - лише вершини в поточному шляху.

Б) У DFS `visited` використовується для циклів, у Гамільтоновому пошуку - для шляхів.

В) У DFS `visited` скидається після кожного рекурсивного виклику, у Гамільтоновому пошуку - ні.

Г) Немає відмінностей, вони використовуються однаково.

5. Теорема Дірака надає:

   А) Необхідні умови для існування Гамільтонового циклу.

   Б) Достатні умови для існування Гамільтонового циклу.

   В) Алгоритм для знаходження Ейлерового циклу.

   Г) Метод перевірки на двочастковість.

6. Якщо зв’язний неорієнтований граф має рівно дві вершини з непарним ступенем, то він:

   А) Має Ейлерів цикл.

   Б) Має Ейлерів шлях, але не цикл.

   В) Не має ні Ейлерового шляху, ні Ейлерового циклу.

   Г) Є Гамільтоновим.

7. Яка з цих задач є типовим застосуванням Гамільтонового циклу?

   А) Пошук шляхів між серверами в мережі.

   Б) Оптимізація послідовності операцій на конвеєрі, де кожна операція повинна бути виконана рівно один раз.

   В) Визначення взаємозв’язків між елементами в базі даних.

   Г) Обчислення потоку в транспортній мережі.

8. Чому важливо використовувати `visited` масив і `path` масив у алгоритмі backtracking для Гамільтонового циклу, а не просто `visited` як у звичайному DFS?

   А) Тому що Гамільтонів цикл може відвідувати вершини кілька разів.

Б) Тому що `path` зберігає фактичну послідовність вершин, а `visited` дозволяє відстежувати унікальність відвідувань у поточному шляху.

В) `path` використовується для відновлення шляху, а `visited` для перевірки зв'язності.

Г) `visited` у цьому випадку не потрібен, достатньо лише `path`.

9. Якщо орієнтований граф має Ейлерів цикл, він обов'язково є:

   А) Ациклічним.

   Б) Двочастковим.

   В) Сильно зв'язним (серед вершин ненульового ступеня).

   Г) Повним.

10. Яка основна відмінність між ефективними алгоритмами для Ейлерових шляхів/циклів та алгоритмами для Гамільтонових шляхів/циклів?

    А) Ейлерові шляхи використовують BFS, а Гамільтонові - DFS.

    Б) Ейлерові шляхи використовують матрицю суміжності, а Гамільтонові - списки суміжності.

    В) Ейлерові шляхи мають поліноміальну часову складність, тоді як Гамільтонові - експоненціальну.

    Г) Ейлерові шляхи застосовні лише до орієнтованих графів, а Гамільтонові - до неорієнтованих.

Відповіді:

1 В). Через часту перевірку на наявність мостів.

2 Б). Існує одна вершина S з `out_degree(S) = in_degree(S) + 1` і одна вершина T з `in_degree(T) = out_degree(T) + 1`, а інші мають рівні ступені.

3 Б). Оптимальний маршрут робота-прибиральника, який має проїхати кожною ділянкою підлоги складу.

4 А). У DFS `visited` відстежує всі відвідані вершини, у Гамільтоновому пошуку - лише вершини в поточному шляху.

5 Б). Достатні умови для існування Гамільтонового циклу.

6 Б). Має Ейлерів шлях, але не цикл.

7 Б). Оптимізація послідовності операцій на конвеєрі, де кожна операція повинна бути виконана рівно один раз.

8 Б). Тому що `path` зберігає фактичну послідовність вершин, а `visited` дозволяє відстежувати унікальність відвідувань у поточному шляху.

9 В). Сильно зв'язним (серед вершин ненульового ступеня).

10 В). Ейлерові шляхи мають поліноміальну часову складність, тоді як Гамільтонові - експоненціальну.

Завдання для самостійної роботи

1. Алгоритм Ієрхольцера: Досліди та реалізуй більш ефективний алгоритм Ієрхольцера для знаходження Ейлерового шляху/циклу. Порівняй його з алгоритмом Флері з точки зору складності та принципу роботи.
2. Ейлерів шлях для орієнтованих графів: Модифікуй свій алгоритм для знаходження Ейлерового шляху/циклу, щоб він працював з орієнтованими графами, враховуючи `in_degree` та `out_degree`.
3. Кількість Гамільтонових циклів: Модифікуй алгоритм пошуку Гамільтонового циклу, щоб він не просто знаходив один цикл, а рахував їхню загальну кількість (без дублікатів).
4. Візуалізація Гамільтонових шляхів: Спробуй додати візуалізацію для знайдених Гамільтонових шляхів/циклів (наприклад, підсвічування вершин та ребер у порядку обходу).
5. Задача комівояжера (TSP): Досліди, що таке Задача комівояжера. Зрозумій, як вона пов'язана з Гамільтоновим циклом. Спробуй реалізувати простий жадібний алгоритм або алгоритм "найближчого сусіда" для наближеного розв'язання TSP.
6. Умови існування Гамільтонового циклу: Досліди теореми Дірака та Оре. Поясни, що вони означають і чому вони є лише достатніми, а не необхідними умовами.

7. Напівейлерові графи: Створи напівейлерів граф (має шлях, але не цикл) і протестуй на ньому свій алгоритм пошуку Ейлерового шляху.
8. Гамільтонів шлях: Модифікуй алгоритм пошуку Гамільтонового циклу так, щоб він знаходив Гамільтонів шлях (який не обов'язково закінчується у початковій вершині).
9. Динамічне програмування для Гамільтонових шляхів: Досліди, як Гамільтонів шлях/цикл можна знайти за допомогою динамічного програмування з використанням бітових масок (для малих графів). Спробуй реалізувати цей підхід.
10. Проблема покриття шляхами: Досліди проблему покриття вершин графа мінімальною кількістю шляхів, що не перетинаються. Як це може бути пов'язано з Гамільтоновими шляхами?

Тем лабораторних робіт

1. Лабораторна робота: Розширений пошук циклів
    - Завдання: Розробити окремі функції для ефективного виявлення наявності циклів у неорієнтованих та орієнтованих графах. Для орієнтованих графів, окрім простого виявлення, спробуй вивести один зі знайдених циклів.
2. Лабораторна робота (дві роботи): Пошук Ейлерових та Гамільтонових Шляхів/Циклів
    - Завдання: Реалізуй алгоритм для перевірки існування та знаходження Ейлерового шляху/циклу. Додатково, досліди (без повної реалізації NP-складної задачі) евристичні підходи або алгоритми для пошуку Гамільтонових шляхів/циклів для малих графів.

Алгоритм Ієрхольцера для знаходження Ейлерових шляхів/циклів

Як ми вже знаємо з попереднього розділу, Ейлерів шлях — це шлях, який проходить через кожне ребро графа рівно один раз. Ейлерів цикл — це Ейлерів шлях, що починається і закінчується в одній і тій же вершині. Ми також вивчили чіткі критерії існування таких шляхів/циклів на основі ступенів вершин.

Алгоритм Флері, який ми розглядали раніше, хоч і концептуально простий, але є неефективним через постійні перевірки на мости.

Алгоритм Ієрхольцера пропонує набагато ефективніший підхід, який працює за лінійний час, тобто за $O(V + E)$, що є оптимальним, оскільки кожну вершину та ребро потрібно відвідати хоча б один раз.

Алгоритм Ієрхольцера базується на ідеї, що Ейлерів цикл можна побудувати, послідовно "зшиваючи" цикли. Якщо граф має Ейлерів цикл, то з будь-якої вершини можна почати обхід, рухаючись по ребрах. Якщо в якийсь момент ми потрапляємо у вершину, з якої немає невикористаних ребер, але є невикористані ребра десь в іншому місці графа, це означає, що ми не знайдемо повного Ейлерового циклу за один прохід.

Ієрхольцер вирішує цю проблему, використовуючи два ключові прийоми:

1. Послідовний пошук циклів: Алгоритм починає з довільної вершини і будує цикл, доки не потрапляє у вершину, з якої не може продовжити рух.

2. Продовження з "незавершеної" вершини: Якщо після знаходження такого циклу залишилися невикористані ребра, алгоритм знаходить першу вершину в щойно побудованому циклі, з якої ще є невикористані ребра, і з цієї вершини починає будувати новий цикл. Цей новий цикл потім "вставляється" у вже існуючий. Цей процес повторюється, доки всі ребра графа не будуть використані.

Цей підхід гарантує, що ми завжди зможемо знайти шлях, оскільки у вершини, що не є початковою/кінцевою, кількість вхідних і вихідних ребер (або просто ступінь) парна, що дозволяє завжди "вийти", якщо ми "ввійшли".

Принципи роботи:

Алгоритм Ієрхольцера працює наступним чином:

1. Попередня перевірка: Перевірити, чи граф відповідає умовам для існування Ейлерового шляху/циклу (зв'язність, парні ступені для циклу, або рівно два непарних ступені для шляху). Якщо ні, завершити роботу, Ейлерового шляху/циклу немає.

2. Визначення стартової вершини:

    - Якщо існує Ейлерів цикл (усі вершини мають парний ступінь), обрати будь-яку вершину як початкову.

- Якщо існує Ейлерів шлях (рівно дві вершини мають непарний ступінь), обрати одну з вершин з непарним ступенем як початкову.

3. Ініціалізація стека: Створити порожній стек для тимчасового зберігання вершин, які ми відвідуємо.
4. Ініціалізація результуючого шляху: Створити порожній список `eulerian_circuit` для зберігання вершин кінцевого Ейлерового шляху/циклу.
5. Основний обхід:
    - Додати початкову вершину до стека.
    - Поки стек не порожній:
        - Нехай `current_vertex` буде вершиною на вершині стека.
        - Якщо з `current_vertex` є невикористані ребра:
            - Обрати будь-яке невикористане ребро $(current_vertex, next_vertex)$.
            - Видалити це ребро з графа (позначити як використане).
            - Додати `next_vertex` до стека.
            - Встановити `current_vertex = next_vertex`.
        - Якщо з `current_vertex` немає невикористаних ребер:
            - Вилучити `current_vertex` зі стека.
            - Додати `current_vertex` до початку списку `eulerian_circuit`. (Або до кінця, а потім перевернути список).
6. Результат: Після завершення, `eulerian_circuit` буде містити Ейлерів шлях/цикл.

Важливо, що “видалення ребра” може бути реалізовано логічно, наприклад, шляхом підтримки лічильників ступенів або масивів `visited_edges`, щоб уникнути зміни початкової структури графа, якщо вона потрібна для інших

цілей. Алгоритм ефективно працює, зменшуючи ступінь вершин при використанні ребер.

Псевдокод алгоритму Ієрхольцера відображає основну логіку для неорієнтованих графів, де всі вершини мають парний ступінь (для циклу).

Структури даних, що використовуються:

- `Graph G`: Вхідний граф (неорієнтований або орієнтований).
- `AdjacencyList G.Adj`: Список суміжності графа. (Для ефективного видалення ребер та їх відстеження, `AdjacencyList` може містити `(neighbor, edge_id)` або бути змінною структурою).
- `current_degrees`: Масив, що зберігає поточні (невикористані) ступені вершин.
- `path_stack`: Стек для тимчасового зберігання вершин під час обходу.
- `eulerian_path_result`: Список для зберігання вершин кінцевого Ейлерового шляху/циклу.

```
// Допоміжні операції:
// ПОРАХУВАТИ_ПОЧАТКОВІ_СТУПЕНІ(Graph G): Обчислює початкові ступені (in/out для орієнтованих) всіх вершин.
// ПЕРЕВІРИТИ_ЧИ_ЗВ'ЯЗНИЙ_НЕНУЛЬОВИЙ(Graph G): Перевіряє зв'язність тільки між вершинами з ненульовим ступенем.
// ДОДАТИ_В_СТЕК(Стек, Елемент): Додає Елемент на вершину Стеку.
// ВИДАЛИТИ_З_СТЕКУ(Стек): Вилучає та повертає елемент з вершини Стеку.
// СТЕК_НЕ_ПОРОЖНІЙ(Стек): Перевіряє, чи Стек містить елементи.
// ДОДАТИ_В_ПОЧАТОК_СПИСКУ(Список, Елемент): Додає Елемент на початок Списку.
// ОТРИМАТИ_СУСІДІВ_ТА_ВИДАЛИТИ_РЕБРО(Graph G, Вершина U):
//   Призначення: Повертає будь-якого сусіда V вершини U і логічно видаляє ребро (U, V).
//   Деталізація: Зменшує ступінь U та V (або in/out-ступінь). Видаляє V з G.Adj[U] та U з G.Adj[V].

АЛГОРИТМ HIERHOLZER_EULERIAN_PATH(Graph G):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. current_degrees = ПОРАХУВАТИ_ПОЧАТКОВІ_СТУПЕНІ(G) // Для неорієнтованих: масив ступенів
  3. СТВОРИТИ КОПІЮ_ГРАФА G_MODIFIABLE = G // Робоча копія графа для видалення ребер
```

```
  4. // Крок 1: Попередня перевірка критеріїв існування
  5. ПОЧАТКОВА_ВЕРШИНА = -1 // Визначення початкової вершини для обхо
ду
  6. КІЛЬКІСТЬ_НЕПАРНИХ_ВЕРШИН = 0

  7. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. ЯКЩО current_degrees[i] % 2 != 0: // Якщо ступінь непарний
          i. КІЛЬКІСТЬ_НЕПАРНИХ_ВЕРШИН = КІЛЬКІСТЬ_НЕПАРНИХ_ВЕРШИН +
1
          ii. ПОЧАТКОВА_ВЕРШИНА = i

  8. ЯКЩО КІЛЬКІСТЬ_НЕПАРНИХ_ВЕРШИН == 0: // Маємо Ейлерів цикл, можн
а почати з будь-якої вершини ненульового ступеня
      a. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
          i. ЯКЩО current_degrees[i] > 0:
              1. ПОЧАТКОВА_ВЕРШИНА = i
              2. ПЕРЕРВАТИ ЦИКЛ
  9. ІНАКШЕ ЯКЩО КІЛЬКІСТЬ_НЕПАРНИХ_ВЕРШИН == 2: // Маємо Ейлерів шля
х, ПОЧАТКОВА_ВЕРШИНА вже визначена
      a. // Все добре, продовжуємо
  10. ІНАКШЕ: // Граф не відповідає критеріям Ейлера
      a. ПОВЕРНУТИ "Ейлерів шлях/цикл не існує"

  11. ЯКЩО ПОЧАТКОВА_ВЕРШИНА == -1: // Граф складається лише з ізольо
ваних вершин або порожній.
      a. ПОВЕРНУТИ "Граф порожній або складається з ізольованих верши
н."

  12. // Перевірка зв'язності (тільки для вершин ненульового ступеня)
  13. ЯКЩО НЕ ПЕРЕВІРИТИ_ЧИ_ЗВ'ЯЗНИЙ_НЕНУЛЬОВИЙ(G_MODIFIABLE, ПОЧАТКО
ВА_ВЕРШИНА):
      a. ПОВЕРНУТИ "Ейлерів шлях/цикл не існує (граф не зв'язний)"

  14. // Крок 2: Побудова шляху
  15. СТВОРИТИ ПОРОЖНІЙ Stack path_stack
  16. СТВОРИТИ ПОРОЖНІЙ СПИСОК eulerian_path_result

  17. current_vertex = ПОЧАТКОВА_ВЕРШИНА
  18. ДОДАТИ_В_СТЕК(path_stack, current_vertex)

  19. ПОКИ СТЕК_НЕ_ПОРОЖНІЙ(path_stack):
      a. U = ОТРИМАТИ_ВЕРШИНУ_З_ВЕРШИНИ_СТЕКА(path_stack) // Просто п
одивитись, не вилучаючи
      b. ЯКЩО current_degrees[U] > 0: // Якщо з U ще є невикористані
```

```
ребра
          i. current_vertex = U
          ii. V = ОТРИМАТИ_СУСІДА_ТА_ВИДАЛИТИ_РЕБРО(G_MODIFIABLE, cur
rent_vertex) // Отримуємо сусіда і видаляємо ребро (current_vertex, V
)
          iii. current_degrees[current_vertex] = current_degrees[curr
ent_vertex] - 1
          iv. current_degrees[V] = current_degrees[V] - 1 // Зменшити
ступінь обох вершин (для неорієнтованого)
          v. ДОДАТИ_В_СТЕК(path_stack, V) // Додаємо наступну вершину
до стека
      c. ІНАКШЕ: // З U немає невикористаних ребер
          i. U_FINISHED = ВИДАЛИТИ_З_СТЕКУ(path_stack) // Вилучаємо U
зі стека
          ii. ДОДАТИ_В_ПОЧАТОК_СПИСКУ(eulerian_path_result, U_FINISHE
D) // Додаємо U до результату

  20. // Остаточна перевірка: чи всі ребра були використані
  21. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. ЯКЩО current_degrees[i] != 0: // Якщо залишились невикориста
ні ребра
          i. ПОВЕРНУТИ "Помилка: Не всі ребра були використані (граф
не Ейлерів). "

  22. ПОВЕРНУТИ eulerian_path_result
```

Операції та їх пояснення (для алгоритму Ієрхольцера):

- **ПОРАХУВАТИ_ПОЧАТКОВІ_СТУПЕНІ(Graph G)**: Ця функція обчислює початкові ступені (для неорієнтованих графів) або вхідні/вихідні ступені (для орієнтованих) для всіх вершин графа G. Це необхідно для попередньої перевірки критеріїв Ейлера.
- **ПЕРЕВІРИТИ_ЧИ_ЗВ'ЯЗНИЙ_НЕНУЛЬОВИЙ(Graph G, ПОЧАТКОВА_ВЕРШИНА)**: Ця допоміжна функція перевіряє, чи є граф (або його частина, що містить ребра) зв’язним. Вона використовує BFS або DFS, але обходить лише ті вершини, які мають ненульовий ступінь, і перевіряє, чи всі вершини з ненульовим ступенем є досяжними з **ПОЧАТКОВОЇ_ВЕРШИНИ**. Це необхідно, щоб уникнути ізольованих ребер або компонент, що не містять початкової вершини, але мають ребра.
- **СТВОРИТИ КОПІЮ_ГРАФА G_MODIFIABLE = G**: Для того, щоб логічно “видаляти” ребра, ми працюємо з копією графа. Це може

бути реалізовано як копія списків суміжності, де ребра можуть бути видалені, або ж можна використовувати матрицю суміжності для відстеження використання ребер.

– `ПОЧАТКОВА_ВЕРШИНА`: Змінна, яка зберігає вершину, з якої ми почнемо будувати наш Ейлерів шлях/цикл. Вибір цієї вершини залежить від попередньої перевірки ступенів.

– `КІЛЬКІСТЬ_НЕПАРНИХ_ВЕРШИН`: Лічильник, який використовується для перевірки критеріїв Ейлера. Для циклу їх має бути 0, для шляху — 2.

– `path_stack`: Це стек, який використовується для тимчасового зберігання вершин під час поточного обходу. Він допомагає відстежувати "поточну траєкторію" обходу.

– `eulerian_path_result`: Це список, який буде містити вершини у порядку їхнього обходу, формуючи кінцевий Ейлерів шлях або цикл. Вершини додаються до нього, коли вони вилучаються зі `path_stack`.

– `ДОДАТИ_В_СТЕК(path_stack, current_vertex)`: Коли ми переходимо до нової вершини, ми "запам'ятовуємо" її, додаючи до стека.

– `ОТРИМАТИ_СУСІДА_ТА_ВИДАЛИТИ_РЕБРО(G_MODIFIABLE, current_vertex)`: Це ключова операція. Вона знаходить невикористане ребро, що виходить з `current_vertex`, повертає сусідню вершину `V` і одночасно логічно "видаляє" це ребро з графа `G_MODIFIABLE`. "Видалення" може означати зменшення лічильника ступеня, видалення зі списку суміжності, або позначення ребра як використаного.

– `current_degrees[U] > 0`: Ця умова перевіряє, чи є ще невикористані ребра, що виходять з вершини `U`.

– `ІНАКШЕ (current_degrees[U] == 0)`: Якщо з вершини `U` немає невикористаних ребер, це означає, що ми застрягли або завершили частину циклу. У цьому випадку:

  - `U_FINISHED = ВИДАЛИТИ_З_СТЕКУ(path_stack)`: Вершина `U` вилучається зі стека.

- `ДОДАТИ_В_ПОЧАТОК_СПИСКУ(eulerian_path_result, U_FINISHED)`: Вершина `U_FINISHED` додається до `eulerian_path_result`. Важливо додавати її до початку списку, оскільки вершини вилучаються зі стека в зворотному порядку до того, як вони були додані. Або можна додавати до кінця, а потім перевернути весь список.

Часова складність: Алгоритм Ієрхольцера має часову складність $O(V + E)$, що є лінійним часом і оптимальним для цієї задачі. Це значно ефективніше, ніж алгоритм Флері, оскільки він не вимагає частих перевірок на мости.

Просторова складність також $O(V + E)$ для зберігання графа та стека/результату.

2. Область застосування

Алгоритм Ієрхольцера, застосовується у всіх сценаріях, де необхідно пройти кожну "доріжку" або "з'єднання" рівно один раз.

- Логістика та планування маршрутів: Найбільш класичне застосування. Це може бути маршрут сміттєвоза, який повинен зібрати сміття з кожної вулиці міста; маршрут снігоприбиральної машини; маршрут патрулювання безпеки, що проходить по всіх об'єктах охорони. Мета — мінімізувати пройдений шлях, уникаючи повторного проїзду тими ж ділянками.

- Сервісне обслуговування: Планування маршрутів обслуговуючого персоналу, який повинен перевірити кожну ділянку мережі (наприклад, електромережі, трубопроводу).

- Виробництво та інспекція: Оптимізація роботи роботизованих систем, які повинні виконувати завдання на кожній ділянці поверхні або кожній лінії продукції.

- Біоінформатика: У збірці геномів, де фрагменти ДНК (ребра) потрібно з'єднати в повну послідовність (шлях/цикл), використовуючи кожен фрагмент рівно один раз.

- Тестування схем: Планування тестів для перевірки всіх з'єднань в електричних схемах.

3. Приклади застосування

– Маршрут поштаря/сміттєвоза: Уявіть, що ви є поштарем у невеликому місті, і вам потрібно доставити пошту на кожну вулицю. Ви хочете пройти кожною вулицею рівно один раз, щоб не витрачати зайвий час та паливо. Мережа вулиць міста — це граф, де перехрестя є вершинами, а вулиці — ребрами. Якщо всі вулиці двосторонні, це неорієнтований граф. Якщо всі вулиці мають парний ступінь (або рівно дві вулиці мають непарний ступінь), алгоритм Ієрхольцера може знайти оптимальний маршрут, який дозволить вам пройти по кожній вулиці лише один раз.

– Інспекція мережі: Оператор мережі повинен перевірити кожен сегмент оптоволоконного кабелю в невеликому кампусі. Кожен сегмент кабелю є ребром, а комутатори або точки з'єднання — вершинами. Якщо оператор починає з центрального комутатора і повинен перевірити всі кабелі рівно один раз, алгоритм Ієрхольцера допоможе спланувати шлях, що мінімізує повторні обходи.

– Проектування маршрутів прибирання: Для роботизованих пилососів або інших прибиральних машин, які працюють на великих плоских поверхнях (складах, аеропортах). Якщо "дороги" або "смуги" є ребрами, а точки повороту – вершинами, машина може використовувати Ейлерів цикл, щоб покрити всю площу ефективно, без пропусків та зайвих повторень.

Переваги та Недоліки

Переваги алгоритму Ієрхольцера:

– Висока ефективність: Часова складність $O(V + E)$, що є лінійним часом і є оптимальною, оскільки кожен елемент графа має бути оброблений принаймні один раз. Це робить його придатним для дуже великих графів.

– Не вимагає перевірки на мости: На відміну від алгоритму Флері, Ієрхольцер не потребує частих (і дорогих) перевірок на мости, що значно підвищує його практичну швидкість.

– Простота концепції: Хоча деталі реалізації можуть бути нюансованими, основна ідея "послідовного зшивання циклів" є доволі інтуїтивною.

– Конструктивний: Алгоритм не просто говорить, чи існує Ейлерів шлях/цикл, а й будує його.

Недоліки алгоритму Ієрхольцера:

– Попередня перевірка критеріїв: Все ще вимагає попередньої перевірки умов існування Ейлерового шляху/циклу (зв'язність та аналіз ступенів вершин). Ця перевірка є необхідною, оскільки алгоритм Ієрхольцера зазвичай припускає, що шлях/цикл існує. Якщо він не існує, алгоритм може повернути неповний шлях або некоректний результат.

– Зміни в структурі графа (логічно): Алгоритм вимагає "видалення" ребер після їх використання. Це означає, що ви повинні працювати з копією графа або використовувати механізм відстеження використаних ребер, що може вимагати додаткової логіки.

– Реалізаційні деталі: У деяких випадках, управління "видаленням" ребер та вибором наступного ребра може вимагати обережної реалізації, особливо для орієнтованих графів або мультиграфів (графів з кількома ребрами між однією парою вершин).

Контрольні Запитання

1. Порівняння з Флері: У чому полягає головна перевага алгоритму Ієрхольцера порівняно з алгоритмом Флері з точки зору часової складності?

2. Основна ідея Ієрхольцера: Опишіть двохетапну основну ідею алгоритму Ієрхольцера для побудови Ейлерового шляху/циклу.

3. Вибір наступного ребра: Як алгоритм Ієрхольцера обирає наступне ребро для переходу, і чи відрізняється це від стратегії Флері?

4. Використання стека: Для чого використовується стек (`path_stack`) в алгоритмі Ієрхольцера?

5. Побудова результату: В якому порядку (по відношенню до стека) вершини додаються до кінцевого `eulerian_path_result` в алгоритмі Ієрхольцера, і чому саме в такому порядку?

6. “Видалення” ребер: Як у алгоритмі Ієрхольцера реалізується “видалення” ребра після його використання? Які наслідки це має для структури графа?
7. Часова складність: Яка часова складність алгоритму Ієрхольцера, і чому вона є оптимальною?
8. Перевірка критеріїв: Чи вимагає алгоритм Ієрхольцера попередньої перевірки критеріїв існування Ейлерового шляху/циклу? Якщо так, то чому?
9. Застосування: Наведіть один практичний приклад, де ефективність алгоритму Ієрхольцера є критично важливою.
10. Адаптація для шляху: Як алгоритм Ієрхольцера адаптується для знаходження Ейлерового шляху (не циклу) у графі, що має рівно дві вершини з непарним ступенем?

Тести

1. Яка часова складність алгоритму Ієрхольцера?

   А) $O(V^2)$

   Б) $O(E^2)$

   В) $O(V + E)$

   Г) $O(V \cdot E)$

2. Яка структура даних є ключовою для тимчасового зберігання вершин під час обходу в алгоритмі Ієрхольцера?

   А) Черга

   Б) Стек

   В) Список

   Г) Хеш-таблиця

3. Що НЕ є перевагою алгоритму Ієрхольцера порівняно з алгоритмом Флері?

   А) Вища ефективність.

Б) Не вимагає перевірки на мости під час обходу.

В) Простіша концепція (уникнення повернення).

Г) Побудова транспонованого графа.

4. В алгоритмі Ієрхольцера вершини додаються до кінцевого результату (`eulerian_path_result`):

   А) У порядку, в якому вони додаються до стека.

   Б) Коли вони вилучаються зі стека.

   В) У випадковому порядку.

   Г) Після завершення всіх DFS-обходів.

5. Для чого використовується попередня перевірка критеріїв Ейлера перед запуском алгоритму Ієрхольцера?

   А) Для визначення початкової вершини.

   Б) Для перевірки наявності циклів.

   В) Щоб переконатися, що Ейлерів шлях/цикл дійсно існує.

   Г) Для оптимізації використання пам’яті.

Відповіді: 1 - В), 2 - Б), 3 - Г),4 - Б), 5 - В).

Тести підвищеної складності

1. Якщо граф має Ейлерів шлях, але не цикл, яку вершину слід обрати як початкову для алгоритму Ієрхольцера?

   А) Будь-яку вершину з парним ступенем.

   Б) Будь-яку вершину з непарним ступенем.

   В) Вершину з найбільшим ступенем.

   Г) Довільну вершину.

2. Чому алгоритм Ієрхольцера вважається “конструктивним” алгоритмом?

   А) Він лише перевіряє існування Ейлерового шляху/циклу.

Б) Він будує граф конденсації.

В) Він фактично будує і виводить Ейлерів шлях/цикл.

Г) Він дозволяє візуалізувати граф.

3. Яким чином алгоритм Ієрхольцера “видаляє” ребра з графа під час обходу?

   А) Фізично видаляє ребра з внутрішньої структури графа.

   Б) Позначає ребра як “використані” або зменшує лічильники ступенів/ребер.

   В) Створює новий граф без використаних ребер на кожному кроці.

   Г) Використовує окремий масив для відстеження ребер, які потрібно пропустити.

4. Уявіть, що ви застосовуєте алгоритм Ієрхольцера до орієнтованого графа. Який критерій ступенів має бути задоволений для існування Ейлерового циклу?

   А) Всі вершини мають парний ступінь.

   Б) Вхідний ступінь кожної вершини дорівнює її вихідному ступеню.

   В) Рівно одна вершина має `out_degree = in_degree + 1`.

   Г) Граф є сильно зв’язним і має лише одну вершину з непарним ступенем.

5. Якщо під час виконання алгоритму Ієрхольцера (`ПОКИ СТЕК_НЕ_ПОРОЖНІЙ(path_stack)`) для поточної вершини `U` виявляється, що `current_degrees[U] == 0`, що робить алгоритм?

   А) Перериває роботу, оскільки знайдено помилку.

   Б) Додає `U` до стека знову.

   В) Вилучає `U` зі стека і додає її до початку результуючого шляху.

   Г) Викликає рекурсивний DFS з `U`.

6. Яка з проблем є типовим застосуванням алгоритму Ієрхольцера?

А) Оптимізація маршруту доставки, щоб відвідати кожне місто один раз.

Б) Планування маршруту робота, який повинен пройти кожною лінією на виробництві рівно один раз.

В) Пошук найкоротшого шляху на карті Google.

Г) Створення розкладу проектних завдань з залежностями.

7. Чому граф повинен бути зв’язним (серед вершин ненульового ступеня) для існування Ейлерового шляху/циклу?

   А) Щоб гарантувати, що всі вершини мають парний ступінь.

   Б) Щоб алгоритм міг досягти всіх ребер.

   В) Щоб уникнути циклів у графі.

   Г) Щоб забезпечити швидке виконання алгоритму.

8. Що станеться, якщо запустити алгоритм Ієрхольцера на графі, який не відповідає критеріям Ейлера, без попередньої перевірки?

   А) Алгоритм завершиться коректно і видасть порожній шлях.

   Б) Алгоритм може застрягнути або повернути неповний шлях.

   В) Алгоритм автоматично виправить граф, щоб він став Ейлеровим.

   Г) Алгоритм видасть помилку компіляції.

9. Яке твердження найкраще описує відношення між алгоритмом Ієрхольцера та алгоритмом Флері?

   А) Ієрхольцер є попередником Флері.

   Б) Ієрхольцер є більш ефективною альтернативою Флері.

   В) Флері використовується для Ейлерових шляхів, а Ієрхольцер - для Ейлерових циклів.

   Г) Вони розв’язують різні задачі.

10. У контексті алгоритму Ієрхольцера, що означає “послідовне зшивання циклів”?

А) З’єднання декількох графів в один великий.

Б) Виявлення окремих циклів у графі та їхнє об’єднання для формування Ейлерового шляху/циклу.

В) Розбиття великого циклу на декілька менших.

Г) Побудова циклів паралельно.

Відповіді:

1 Б). Будь-яку вершину з непарним ступенем.

2 В). Він фактично будує і виводить Ейлерів шлях/цикл.

3 Б). Позначає ребра як “використані” або зменшує лічильники ступенів/ребер.

4 Б). Вхідний ступінь кожної вершини дорівнює її вихідному ступеню.

5 В). Вилучає `U` зі стека і додає її до початку результуючого шляху.

6 Б). Планування маршруту робота, який повинен пройти кожною лінією на виробництві рівно один раз.

7 Б). Щоб алгоритм міг досягти всіх ребер.

8 Б). Алгоритм може застрягнути або повернути неповний шлях.

9 Б). Ієрхольцер є більш ефективною альтернативою Флері.

10 Б). Виявлення окремих циклів у графі та їхнє об’єднання для формування Ейлерового шляху/циклу.

Завдання для самостійної роботи

1. Реалізація видалення ребер: Реалізуй метод `ОТРИМАТИ_СУСІДА_ТА_ВИДАЛИТИ_РЕБРО` таким чином, щоб він ефективно видаляв ребро зі списків суміжності для неорієнтованого графа.
2. Ейлерів шлях (не цикл) для Ієрхольцера: Адаптуй алгоритм Ієрхольцера для знаходження Ейлерового шляху, коли граф має рівно дві вершини з непарним ступенем. Переконайся, що алгоритм починає з однієї з них.

3. Орієнтовані графи: Модифікуй алгоритм Ієрхольцера для роботи з орієнтованими графами, враховуючи `in_degree` та `out_degree` для критеріїв існування та для відстеження невикористаних ребер.
4. Мультиграфи: Розглянь, як алгоритм Ієрхольцера працює з мультиграфами (графами, де між двома вершинами може бути кілька ребер). Адаптуй свою реалізацію для їхньої підтримки.
5. Пошук циклів: Вбудуй логіку пошуку циклів (як обговорювалося в попередніх розділах) безпосередньо в алгоритм Ієрхольцера для орієнтованих графів, щоб він міг виявляти, чи не є граф DAG.
6. Відновлення шляху: Модифікуй алгоритм таким чином, щоб він не просто виводив послідовність вершин, а й явно вказував, яке ребро було використане на кожному кроці.
7. Головоломка “Провести лінію”: Розглянь класичну головоломку “намалюй фігуру, не відриваючи олівця від паперу і не проводячи по лінії двічі”. Створи граф, що відповідає цій фігурі, і використай алгоритм Ієрхольцера для її розв’язання.
8. Оптимізація обходу: Якщо граф дуже великий, і його неможливо вмістити повністю в пам’ять, досліди, як алгоритм Ієрхольцера (або його варіації) може бути застосований для обходу зовнішніх графів або графів, що зберігаються на диску.
9. Візуалізація: Створи простий інструмент для візуалізації роботи алгоритму Ієрхольцера: показуй граф, підсвічуй вибрані ребра та вершини, та оновлюй поточний шлях.
10. Проблема мостових об’єктів: Досліди, як алгоритми Ейлерових шляхів застосовуються в контексті 3D-друку для оптимізації руху екструдера, щоб мінімізувати “порожні” рухи.

Найкоротші шляхи, потоки та паросполучення

У цьому розділі ми розглянемо три фундаментальні класи задач комбінаторної оптимізації на графах: найкоротші шляхи, максимальні потоки та паросполучення. Незалежно від конкретної області застосування — від транспортного планування та телекомунікацій до біоінформатики та соціальних мереж — вони дозволяють формулювати та ефективно вирішувати найрізноманітніші практичні завдання.

Спочатку ми детально вивчимо задачу пошуку найкоротших шляхів, яка полягає у визначенні найменшої за вагою або довжиною послідовності ребер між двома заданими вершинами. Типові алгоритми (Дейкстри, Беллмана–Форда, SPFA алгоритм, алгоритм Флойда–Уоршелла, інші) забезпечують рішення зі строгими оцінками складності, що робить їх потужним інструментом для аналізу великих графових моделей.

Далі переходимо до задачі максимального потоку, в якій потрібно знайти найбільший можливий «потік» між джерелом і стоком у спрямованому мережевому графі з обмеженнями на ємності ребер. Алгоритми Едмондса–Карпа, Форда–Фалкерсона та покращені варіанти з використанням розріджених структур даних лежать в основі ефективного управління ресурсами та оптимізації пропускної здатності мереж.

Нарешті, ми розглянемо паросполучення — вибір набору невзаємопересічних ребер у неспрямованому графі так, щоб максимізувати кількість з’єднаних пар вершин. Методи, які спираються на ідеї доповнення шляхів та дипольних структур (алгоритм Хопкрофта–Карпа, теорема Кеніга), застосовують у складанні розкладу, розподілі завдань та багатьох інших задачах дискретної математики.

Цей розділ не лише надасть формальні визначення та опише класичні алгоритмічні підходи, а й продемонструє взаємозв’язок між цими трьома темами, їхні спільні техніки та прикладні кейси.

Найкоротші шляхи (одна вершина)

Алгоритм Дейкстри для знаходження найкоротших шляхів

1. Алгоритм Дейкстри — це жадібний алгоритм, призначений для знаходження найкоротших шляхів від однієї початкової вершини до всіх інших вершин у графі. Він був розроблений Едсгером Дейкстрою у 1956

році. Цей алгоритм є одним з найвідоміших і найчастіше використовуваних алгоритмів у теорії графів.

1.1. Проблема найкоротшого шляху

У попередніх розділах ми вже розглядали поняття шляху в графі. Тепер ми додаємо концепцію ваги до ребер.

- Зважений граф: Це граф, де кожному ребру присвоєно числове значення, яке називається вагою або вартістю. Вага може представляти відстань, час, вартість, пропускну здатність тощо.
- Вага шляху: Вага шляху — це сума ваг усіх ребер, що входять до цього шляху.
- Найкоротший шлях: Найкоротший шлях між двома вершинами — це шлях, який має мінімальну загальну вагу.

Алгоритм Дейкстри вирішує проблему одного джерела найкоротших шляхів (Single-Source Shortest Path, SSSP) для графів з невід'ємними вагами ребер. Це означає, що вага кожного ребра в графі повинна бути більшою або рівною нулю ($w(u, v) \geq 0$). Якщо граф містить ребра з від'ємною вагою, алгоритм Дейкстри може дати некоректні результати (для таких випадків використовується алгоритм Беллмана-Форда, який ми розглянемо пізніше).

Алгоритм Дейкстри працює за жадібним принципом: на кожному кроці він обирає ту вершину, яка ще не була повністю оброблена, і до якої знайдено найкоротший шлях з початкової вершини. Ця вершина вважається "найближчою" серед усіх необроблених.

Для ефективної реалізації цього жадібного вибору алгоритм використовує чергу з пріоритетом (Priority Queue). Черга з пріоритетом — це структура даних, яка дозволяє швидко вилучати елемент з найвищим (або найнижчим) пріоритетом. У випадку алгоритму Дейкстри, пріоритетом є поточна оцінка найкоротшого шляху до вершини.

Основна операція в алгоритмі Дейкстри називається релаксацією ребра.

- Релаксація ребра (U, V): Це процес оновлення оцінки найкоротшого шляху до вершини V, якщо знайдено коротший шлях до V через вершину U.
    - Припустимо, у нас є поточна оцінка найкоротшого шляху від початкової вершини S до U, позначена як `distance[U]`.

- Якщо ми можемо дістатися до V з U по ребру (U, V) з вагою $w(U, V)$, то потенційний шлях до V через U матиме вагу `distance[U] + w(U, V)`.
- Якщо ця нова потенційна вага менша за поточну оцінку найкоротшого шляху до V (`distance[V]`), то ми оновлюємо `distance[V]` на `distance[U] + w(U, V)`. Це означає, що ми знайшли коротший шлях до V. Ми також запам'ятовуємо, що шлях до V тепер проходить через U (зберігаючи `predecessor[V] = U`).

1.2. Покроковий опис алгоритму Дейкстри

1. Ініціалізація:
   - Створити масив `distance` (або `dist`) розміром V, де `distance[i]` зберігатиме поточну оцінку найкоротшого шляху від початкової вершини до вершини i. Ініціалізувати `distance[start_node] = 0` (шлях до самої себе нульовий) і `distance[i] = НЕСКІНЧЕННІСТЬ` для всіх інших вершин (оскільки шляхи до них ще невідомі).
   - Створити масив `predecessor` (або `prev`) розміром V, де `predecessor[i]` зберігатиме попередню вершину на найкоротшому шляху до i. Ініціалізувати всі як `NULL` або `-1`.
   - Створити чергу з пріоритетом `PQ`. Вона буде зберігати пари `(відстань, вершина)`, де пріоритетом є відстань. Додати до `PQ` початкову вершину: `(0, start_node)`.
   - (Опціонально) Створити масив `visited` (або `finalized`) розміром V, ініціалізувати всі як `false`. Цей масив відстежує вершини, для яких найкоротший шлях вже остаточно визначений.
2. Основний цикл:
   - Поки `PQ` не порожня:
     - Вилучити з `PQ` пару `(d, U)` з найменшою відстанню `d`. Це означає, що U є найближчою необробленою вершиною.

        - Якщо `d > distance[U]`, пропустити цю ітерацію. Це відбувається, якщо ми вже знайшли коротший шлях до U раніше (черга з пріоритетом може містити застарілі записи).
        - (Опціонально) Якщо `visited[U]` вже `true`, пропустити цю ітерацію. Це означає, що найкоротший шлях до U вже визначений.
        - (Опціонально) Позначити `visited[U] = true`.
    3. Релаксація сусідів:
        - Для кожної сусідньої вершини V з U (тобто, для кожного ребра (U, V) з вагою $w(U, V)$):
            - Якщо `distance[U] + w(U, V) < distance[V]` (знайдено коротший шлях до V через U):
                - Оновити `distance[V] = distance[U] + w(U, V)`.
                - Оновити `predecessor[V] = U`.
                - Додати `(distance[V], V)` до `PQ`.
    4. Завершення: Коли `PQ` стає порожньою, `distance` масив містить найкоротші відстані від `start_node` до всіх досяжних вершин, а `predecessor` масив дозволяє відновити ці шляхи.

1.3. Псевдокод алгоритму Дейкстри

Структури даних, що використовуються:

- `Graph G`: Вхідний зважений граф з невід'ємними вагами ребер.
- `AdjacencyList G.Adj`: Список суміжності графа, де кожне ребро зберігає вагу (наприклад, `G.Adj[U]` містить пари `(V, w(U,V))`).
- `start_node`: Початкова вершина.
- `distance`: Масив розміром `V`, `distance[i]` зберігає поточну мінімальну відстань від `start_node` до `i`.
- `predecessor`: Масив розміром `V`, `predecessor[i]` зберігає попередню вершину на найкоротшому шляху до `i`.

- **`PriorityQueue PQ`**: Черга з пріоритетом, що зберігає пари **`(відстань, вершина)`**. Елементи вилучаються за мінімальною відстанню.

```
// Допоміжні операції:

// Операція: КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)
// Призначення: Повертає загальну кількість вершин у Графі.

// Операція: ІНІЦІАЛІЗУВАТИ_МАСИВ(Масив, Значення)
// Призначення: Встановлює всі елементи Масиву на задане Значення.

// Операція: ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(PQ, Елемент)
// Призначення: Додає Елемент до черги з пріоритетом.
// Деталізація: Елемент - це пара (пріоритет, значення).

// Операція: ВИЛУЧИТИ_МІНІМАЛЬНИЙ_З_ЧЕРГИ_З_ПРІОРИТЕТОМ(PQ)
// Призначення: Вилучає та повертає елемент з найменшим пріоритетом з черги.
// Повертає: Пару (пріоритет, значення).

// Операція: ЧЕРГА_З_ПРІОРИТЕТОМ_НЕ_ПОРОЖНЯ(PQ)
// Призначення: Перевіряє, чи черга з пріоритетом містить елементи.
// Повертає: true, якщо PQ не порожня, false інакше.

// Операція: ОТРИМАТИ_СУСІДІВ_ТА_ВАГИ(Graph G, Вершина U)
// Призначення: Повертає список пар (сусідня_вершина, вага_ребра) для всіх вихідних ребер з Вершини U.
// Деталізація: Для списків суміжності - G.Adj[U] містить такі пари.

// Константа для нескінченності (дуже велике число)
КОНСТАНТА НЕСКІНЧЕННІСТЬ = 999999999



АЛГОРИТМ DIJKSTRA(Graph G, start_node):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ МАСИВ distance РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  3. СТВОРИТИ МАСИВ predecessor РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  4. СТВОРИТИ PriorityQueue PQ

  5. ІНІЦІАЛІЗУВАТИ_МАСИВ(distance, НЕСКІНЧЕННІСТЬ) // Всі відстані с початку нескінченні
  6. ІНІЦІАЛІЗУВАТИ_МАСИВ(predecessor, NULL) // Попередників немає
```

```
  7. distance[start_node] = 0 // Відстань до початкової вершини - 0
  8. ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(PQ, (0, start_node)) // Додаємо поч
аткову вершину в PQ

  9. ПОКИ ЧЕРГА_З_ПРІОРИТЕТОМ_НЕ_ПОРОЖНЯ(PQ):
      a. (d, U) = ВИЛУЧИТИ_МІНІМАЛЬНИЙ_З_ЧЕРГИ_З_ПРІОРИТЕТОМ(PQ) // В
илучаємо вершину U з найменшою відстанню d

      b. ЯКЩО d > distance[U]: // Якщо ми вже знайшли коротший шлях д
о U
          i. ПРОДОВЖИТИ // Пропускаємо, оскільки це застарілий запис
у черзі

      c. // Релаксація сусідів U
      d. ДЛЯ КОЖНОЇ ПАРИ (V, w_uv) З ОТРИМАТИ_СУСІДІВ_ТА_ВАГИ(G, U):
// Для кожного ребра (U, V) з вагою w_uv
          i. ЯКЩО distance[U] + w_uv < distance[V]: // Якщо знайдено
коротший шлях до V через U
              1. distance[V] = distance[U] + w_uv // Оновити відстань
до V
              2. predecessor[V] = U // Встановити U як попередника V
              3. ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(PQ, (distance[V], V)) /
/ Додати V в PQ з новою відстанню

  10. ПОВЕРНУТИ distance, predecessor // Повертаємо масиви відстаней
та попередників
```

Операції та їх пояснення:

- **КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)**: Ця операція повертає загальну кількість вершин у графі G. Це необхідно для ініціалізації масивів **distance** та **predecessor** відповідного розміру.
- **ІНІЦІАЛІЗУВАТИ_МАСИВ(Масив, Значення)**: Допоміжна операція, яка заповнює всі елементи заданого **Масиву** вказаним **Значенням**. У цьому алгоритмі використовується для встановлення початкових відстаней як **НЕСКІНЧЕННІСТЬ** та попередників як **NULL**.
- **НЕСКІНЧЕННІСТЬ**: Це символічна константа, що представляє дуже велике число, яке більше за будь-яку можливу суму ваг шляху в графі. Вона використовується для ініціалізації відстаней до невідомих вершин.

– `distance[start_node] = 0`: Відстань від початкової вершини до самої себе завжди дорівнює нулю. Це є відправною точкою для алгоритму.

– `ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(PQ, (0, start_node))`: Початкова вершина додається до черги з пріоритетом з пріоритетом 0. Це гарантує, що вона буде першою оброблена.

– `ПОКИ ЧЕРГА_З_ПРІОРИТЕТОМ_НЕ_ПОРОЖНЯ(PQ)`: Це основний цикл алгоритму. Він продовжується до тих пір, поки в черзі є вершини, які потрібно обробити. Коли черга порожня, це означає, що найкоротші шляхи до всіх досяжних вершин були знайдені.

– `(d, U) = ВИЛУЧИТИ_МІНІМАЛЬНИЙ_З_ЧЕРГИ_З_ПРІОРИТЕТОМ(PQ)`: На кожній ітерації з черги вилучається вершина U, яка має найменшу поточну оцінку відстані `d`. Завдяки властивостям черги з пріоритетом, це гарантує, що U є найближчою необробленою вершиною, і шлях до неї вже є найкоротшим можливим.

– `ЯКЩО d > distance[U]: ПРОДОВЖИТИ`: Це важлива оптимізація. Оскільки черга з пріоритетом може містити кілька записів для однієї й тієї ж вершини (якщо її відстань була оновлена кілька разів), цей крок перевіряє, чи поточний вилучений запис `(d, U)` є застарілим. Якщо `d` (відстань у вилученому записі) більша за `distance[U]` (поточну, вже оновлену найкоротшу відстань до U), це означає, що ми вже знайшли коротший шлях до U і обробили його, тому цей застарілий запис можна пропустити.

– `ДЛЯ КОЖНОЇ ПАРИ (V, w_uv) З ОТРИМАТИ_СУСІДІВ_ТА_ВАГИ(G, U)`: Цей цикл перебирає всіх сусідів V поточної вершини U та відповідні ваги ребер w_{uv}.

– `ЯКЩО distance[U] + w_uv < distance[V]`: Це умова релаксації. Вона перевіряє, чи шлях до V через U (`distance[U] + w_uv`) є коротшим за поточну відому найкоротшу відстань до V (`distance[V]`).

– `distance[V] = distance[U] + w_uv`: Якщо знайдено коротший шлях, ми оновлюємо `distance[V]` новим, меншим значенням.

– `predecessor[V] = U`: Ми запам'ятовуємо, що для досягнення V найкоротшим шляхом, ми прийшли з вершини U. Це дозволяє нам відновити сам шлях, рухаючись від кінцевої вершини назад до початкової.

– `ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(PQ, (distance[V], V))`: Після оновлення відстані до V, ми додаємо V до черги з пріоритетом з її новою, меншою відстанню. Це гарантує, що V буде розглянута пізніше, коли настане її черга як найближчої необробленої вершини.

1.4. Часова та просторова складність

Часова складність: Часова складність алгоритму Дейкстри залежить від реалізації черги з пріоритетом.

– Якщо черга з пріоритетом реалізована за допомогою бінарної купи (binary heap):

  - Кожна операція `ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ` (вставка) займає $O(\log V)$ часу.
  - Кожна операція `ВИЛУЧИТИ_МІНІМАЛЬНИЙ_З_ЧЕРГИ_З_ПРІОРИТЕТОМ` займає $O(\log V)$ часу.
  - Ми виконуємо `V` операцій вилучення (по одному для кожної вершини).
  - Ми виконуємо до `E` операцій додавання (по одному для кожного ребра, оскільки кожне ребро може призвести до релаксації).
  - Загальна часова складність: $O((V + E)\log V)$. Для розріджених графів (де $E \approx V$), це приблизно $O(V \log V)$. Для щільних графів (де $E \approx V^2$), це ближче до $O(V^2 \log V)$.

– Якщо черга з пріоритетом реалізована за допомогою фібоначчієвої купи (Fibonacci heap) (складніша реалізація, рідко використовується на практиці через високі постійні фактори):

  - Часова складність: $O(E + V \log V)$. Це теоретично краще для дуже щільних графів.

Просторова складність: Просторова складність алгоритму Дейкстри становить $O(V + E)$ для зберігання графа (списків суміжності), масивів

`distance` та `predecessor`, а також черги з пріоритетом (яка може містити до $O(E)$ елементів у найгіршому випадку, якщо багато ребер призводять до оновлень).

2. Область застосування

Алгоритм Дейкстри є одним з найважливіших алгоритмів у комп'ютерних науках завдяки його здатності ефективно знаходити найкоротші шляхи. Його застосування охоплює широкий спектр галузей:

1. Навігаційні системи:
    - GPS-навігатори: Знаходять найкоротші маршрути між двома точками на карті (міста, вулиці). Міста/перехрестя є вершинами, дороги — ребрами, а відстані/час у дорозі — вагами ребер.
    - Картографічні сервіси: Розрахунок маршрутів для автомобілів, громадського транспорту, пішоходів.
2. Мережева маршрутизація:
    - Протоколи маршрутизації: У комп'ютерних мережах (наприклад, Інтернет), алгоритм Дейкстри або його варіації використовуються для визначення найефективніших шляхів для пакетів даних від джерела до призначення. Маршрутизатори використовують його для побудови таблиць маршрутизації.
    - Оптоволоконні мережі: Оптимізація шляхів для передачі даних.
3. Логістика та транспорт:
    - Планування доставки: Визначення найкоротших або найшвидших маршрутів для кур'єрських служб, вантажоперевезень.
    - Управління ланцюгами поставок: Оптимізація переміщення товарів між складами та точками продажу.
4. Ігри:

- Штучний інтелект (AI): Пошук шляхів для персонажів у комп’ютерних іграх (наприклад, найкоротший шлях до цілі, обхід перешкод).
- Генерація рівнів: Побудова доступних шляхів у процедурно згенерованих лабіринтах.

5. Телекомунікації:
    - Проектування мереж: Визначення оптимального розташування комунікаційних ліній.
    - Оптимізація трафіку: Перенаправлення трафіку для уникнення перевантажень.
6. Біоінформатика:
    - Аналіз біологічних мереж: Знаходження найкоротших шляхів у мережах взаємодії білків або метаболічних шляхів.
7. Фінанси:
    - Арбітраж: Пошук найкоротших шляхів у графах валютних обмінів для виявлення можливостей арбітражу (хоча для цього потрібен алгоритм, що працює з від’ємними вагами, якщо комісії представлені від’ємними значеннями).
8. Приклади застосування

– GPS-навігатор: Коли ви вводите пункт призначення у свій GPS-навігатор, він використовує алгоритм, подібний до Дейкстри. Карта міста моделюється як граф, де перехрестя є вершинами, а ділянки доріг між ними — ребрами. Вага кожного ребра може бути відстанню цієї ділянки дороги, або часом, необхідним для її проїзду (з урахуванням обмежень швидкості, світлофорів, заторів). Алгоритм Дейкстри швидко знаходить найкоротший (або найшвидший) шлях від вашого поточного місцезнаходження до пункту призначення.

– Маршрутизація пакетів в Інтернеті: Коли ви надсилаєте дані через Інтернет (наприклад, завантажуєте веб-сторінку), ці дані розбиваються на невеликі пакети. Ці пакети подорожують через мережу маршрутизаторів. Кожен маршрутизатор має таблицю маршрутизації, яка вказує, куди надсилати наступний пакет, щоб він

досяг кінцевого пункту призначення найефективнішим шляхом. Алгоритми, подібні до Дейкстри (наприклад, OSPF - Open Shortest Path First), використовуються маршрутизаторами для постійного оновлення цих таблиць, знаходячи найкоротші шляхи до всіх інших маршрутизаторів у мережі, де "вагою" ребра може бути затримка, пропускна здатність або вартість передачі.

– Пошук шляху для AI у відеоіграх: У багатьох відеоіграх, особливо стратегіях або RPG, персонажі, контрольовані комп'ютером (AI), повинні знаходити шляхи до певних точок на карті, уникаючи перешкод. Карта гри може бути представлена як сітка або граф, де клітинки/області є вершинами, а можливі переходи між ними — ребрами. Ваги ребер можуть відображати складність переміщення по певній місцевості (наприклад, пересування по болоту дорожче, ніж по дорозі). Алгоритм Дейкстри дозволяє AI ефективно знаходити найкоротший шлях до цілі.

Переваги та Недоліки

Переваги алгоритму Дейкстри:

– Гарантія оптимальності: Якщо всі ваги ребер невід'ємні, алгоритм Дейкстри гарантовано знаходить найкоротший шлях.

– Універсальність: Застосовний як до орієнтованих, так і до неорієнтованих графів (для неорієнтованих графів кожне неорієнтоване ребро (U, V) з вагою w можна розглядати як два орієнтованих ребра (U, V) і (V, U) з вагою w).

– Ефективність: З використанням черги з пріоритетом на основі бінарної купи, він є дуже ефективним для більшості практичних задач.

– Один джерело до всіх: Знаходить найкоротші шляхи не лише до однієї цільової вершини, а й до всіх інших досяжних вершин від початкового джерела. Це дозволяє повторно використовувати результати для різних цільових вершин.

Недоліки алгоритму Дейкстри:

– Не працює з від'ємними вагами ребер: Це найсуттєвіший недолік. Якщо граф містить ребра з від'ємною вагою, алгоритм може дати неправильні результати. Це пов'язано з його жадібною природою —

він “фіналізує” шлях до вершини, припускаючи, що подальші релаксації не знайдуть коротшого шляху через від’ємні цикли.

- Може бути повільнішим за BFS для незважених графів: Для незважених графів (де всі ваги ребер дорівнюють 1), BFS знаходить найкоротші шляхи за $O(V + E)$ часу, що є швидше, ніж $O((V + E)\log V)$ у Дейкстри. В таких випадках BFS є кращим вибором.
- Потребує черги з пріоритетом: Хоча це і є перевагою для ефективності, реалізація черги з пріоритетом може бути складнішою, ніж проста черга для BFS.

Контрольні Запитання

1. Призначення Дейкстри: Яку проблему вирішує алгоритм Дейкстри?
2. Обмеження ваг: Яке ключове обмеження на ваги ребер існує для коректної роботи алгоритму Дейкстри? Чому це обмеження важливе?
3. Жадібний підхід: Поясніть, чому алгоритм Дейкстри вважається “жадібним”.
4. Черга з пріоритетом: Яка структура даних є фундаментальною для ефективної реалізації алгоритму Дейкстри, і чому?
5. Релаксація ребра: Опишіть процес “релаксації” ребра (U, V) в алгоритмі Дейкстри. Що відбувається, коли ребро релаксується?
6. Ініціалізація: Як ініціалізуються масиви `distance` та `predecessor` на початку алгоритму Дейкстри?
7. Застарілі записи: Навіщо в основному циклі алгоритму Дейкстри (після вилучення `(d, U)` з PQ) перевіряється умова `d > distance[U]`?
8. Відновлення шляху: Як можна відновити сам найкоротший шлях від початкової вершини до цільової, використовуючи результати алгоритму Дейкстри?
9. Часова складність: Яка часова складність алгоритму Дейкстри з використанням бінарної купи?

10. Порівняння з BFS: Чому BFS є кращим вибором для знаходження найкоротших шляхів у незважених графах, ніж алгоритм Дейкстри?

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота: Реалізація алгоритму Дейкстри для пошуку найкоротших шляхів

2. Мета

Метою даної лабораторної роботи є набуття практичних навичок у реалізації алгоритму Дейкстри для знаходження найкоротших шляхів у зважених графах з невід'ємними вагами ребер, а також розуміння його застосування та обмежень.

3. Завдання

a) Реалізація представлення зваженого графа: Створити клас або структуру для представлення орієнтованого зваженого графа за допомогою списків суміжності. Кожне ребро повинно мати вагу.

b) Реалізація черги з пріоритетом: Створити власну просту реалізацію черги з пріоритетом (наприклад, на основі списку, який сортується при вставці/вилученні, або використовуючи вбудовані структури, якщо дозволено).

c) Реалізація алгоритму Дейкстри: Розробити функцію `dijkstra(Graph G, int start_node)`, яка повертає масиви найкоротших відстаней та попередників.

d) Відновлення шляху: Реалізувати функцію `reconstructPath(int[] predecessor, int start_node, int target_node)`, яка відновлює та виводить найкоротший шлях від початкової до цільової вершини.

e) Тестування: Протестувати реалізований алгоритм на різних зважених графах, включаючи випадки, де:

- Граф є зв'язним.
- Граф є незв'язним (деякі вершини недосяжні).

- Граф містить ребра з різними невід'ємними вагами.
- (Додатково) Продемонструвати некоректну роботу на графі з від'ємною вагою ребра.

4. Опис

Вам необхідно розробити програмний додаток на обраній мові програмування (наприклад, Python, C++, Java, JavaScript). Додаток повинен дозволяти:

– Створювати орієнтовані зважені графи з заданою кількістю вершин та ребер.

– Додавати ребра з вагами до графа.

– Викликати функцію **`dijkstra`** для знаходження найкоротших шляхів від заданої початкової вершини.

– Виводити знайдені найкоротші відстані до всіх вершин.

– Виводити відновлений найкоротший шлях до заданої цільової вершини.

Використання списків суміжності для представлення графа є обов'язковим.

6. Порядок Виконання

a) Підготовка:

   a. Створіть новий проект або файл для лабораторної роботи.

   b. Визначте, як будете представляти зважені орієнтовані графи (наприклад, **`Map<Integer, List<Pair<Integer, Integer>>>`** де **`Pair`** це **`(сусід, вага)`**).

b) Реалізація черги з пріоритетом:

   a. Якщо ваша мова програмування не має вбудованої черги з пріоритетом, реалізуйте її (наприклад, використовуючи **`java.util.PriorityQueue`** в Java, **`heapq`** в Python, або власну реалізацію на основі масиву/списку). Важливо, щоб вона могла зберігати пари **`(пріоритет, елемент)`** і швидко вилучати елемент з мінімальним пріоритетом.

c) Реалізація алгоритму Дейкстри:

a. Створіть функцію `dijkstra(Graph G, int start_node)`.

b. Ініціалізуйте `distance` масив (з `НЕСКІНЧЕННІСТЬ`) та `predecessor` масив (з `NULL`).

c. Встановіть `distance[start_node] = 0` та додайте `(0, start_node)` до PQ.

d. Реалізуйте основний цикл `ПОКИ PQ НЕ ПОРОЖНЯ`, включаючи вилучення мінімального елемента, перевірку на застарілі записи та релаксацію сусідів.

d) Реалізація відновлення шляху:

a. Створіть функцію `reconstructPath(int[] predecessor, int start_node, int target_node)`.

b. Ця функція повинна рухатися назад від `target_node` до `start_node` за допомогою масиву `predecessor` і зібрати шлях.

e) Тестування:

a. Створіть кілька екземплярів `Graph` для різних тестових сценаріїв.

b. Для кожного графа:

i. Викличте `dijkstra` з різними `start_node`.

ii. Виведіть масив `distance`.

iii. Викличте `reconstructPath` для кількох `target_node` і виведіть відновлені шляхи.

c. Обов'язково: Створіть один тестовий граф з від'ємною вагою ребра і продемонструйте, що Дейкстра дає некоректний результат (наприклад, шляхом показу неоптимального шляху або нескінченного циклу, якщо є від'ємний цикл).

f) Результат

Після виконання всіх завдань програми повинні виводити:

– Для кожного тестового графа та початкової вершини:

- Список найкоротших відстаней від початкової вершини до всіх інших вершин (наприклад, `Відстані від 0: {0: 0, 1: 5, 2: 8, 3: INF}`).
- Для кожної заданої цільової вершини: відновлений найкоротший шлях (наприклад, `Шлях до 2: 0 -> 1 -> 2`).

– Для графа з від'ємною вагою: Чітке повідомлення про некоректність результату Дейкстри та демонстрація цього.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:
   - Реалізовано структуру даних для представлення орієнтованого зваженого графа за допомогою списків суміжності.
   - Надано функціональну реалізацію алгоритму Дейкстри.
   - Надано функцію для відновлення найкоротшого шляху.
   - Код має бути добре структурованим, читабельним та містити необхідні коментарі, що пояснюють ключові частини реалізації (наприклад, логіку черги з пріоритетом, релаксацію).
2. Продемонструвати роботу додатку:
   - Продемонструвати роботу алгоритму Дейкстри на 2-3 прикладах графів, наданих викладачем, та на 1-2 власних тестових графах, включаючи випадки з недосяжними вершинами.
   - Показати коректність виводу найкоротших відстаней та відновлених шляхів.
   - Обов'язково продемонструвати приклад графа з від'ємною вагою ребра, де алгоритм Дейкстри дає некоректний результат, і пояснити, чому так відбувається.
3. Бути готовим відповісти на питання:
   - Принцип роботи: Детально пояснити покрокову роботу алгоритму Дейкстри, включаючи ініціалізацію, вибір вершини з черги з пріоритетом та процес релаксації.

- Обмеження: Чому алгоритм Дейкстри не працює з від'ємними вагами ребер? Наведіть приклад, що ілюструє це.
- Складність: Обґрунтувати часову та просторову складність реалізованої версії алгоритму Дейкстри.
- Оптимізація: Пояснити, як використання черги з пріоритетом впливає на ефективність алгоритму порівняно з простою чергою.
- Відновлення шляху: Пояснити логіку відновлення шляху за допомогою масиву `predecessor`.
- Застосування: Навести 2-3 приклади реальних задач, де алгоритм Дейкстри є ключовим.

Тести

1. Алгоритм Дейкстри знаходить найкоротші шляхи в графах з:

   А) Будь-якими вагами ребер.

   Б) Тільки від'ємними вагами ребер.

   В) Тільки невід'ємними вагами ребер.

   Г) Тільки нульовими вагами ребер.

2. Яка структура даних є фундаментальною для ефективної реалізації алгоритму Дейкстри?

   А) Стек.

   Б) Звичайна черга (FIFO).

   В) Черга з пріоритетом.

   Г) Хеш-таблиця.

3. Операція "релаксації" ребра (U, V) означає:

   А) Видалення ребра (U, V) з графа.

   Б) Оновлення ваги ребра (U, V).

   В) Оновлення оцінки найкоротшого шляху до V, якщо знайдено коротший шлях через U.

Г) Позначення вершини U як відвіданої.

4. Яке початкове значення присвоюється відстані до початкової вершини в алгоритмі Дейкстри?

   А) Нескінченність.

   Б) 1.

   В) 0.

   Г) -1.

5. Якщо граф не має від'ємних ваг ребер, але є незв'язним, що буде значенням `distance[V]` для вершини V, недосяжної з початкової вершини?

   А) 0.

   Б) Нескінченність.

   В) Вага найдовшого шляху.

   Г) Вага найкоротшого шляху до найближчої досяжної вершини.

Відповіді: 1 - В), 2 - В),3 - В), 4 - В),5 - Б).

Тести підвищеної складності

1. Чому алгоритм Дейкстри може дати некоректні результати, якщо граф містить від'ємні ваги ребер?

   А) Він не може обробляти від'ємні числа.

   Б) Його жадібний вибір може "фіналізувати" шлях до вершини, який пізніше виявиться неоптимальним через від'ємний цикл.

   В) Черга з пріоритетом не працює з від'ємними числами.

   Г) Він завжди знаходить найдовший шлях у таких графах.

2. Яка часова складність алгоритму Дейкстри, якщо черга з пріоритетом реалізована за допомогою бінарної купи?

   А) $O(V^2)$

   Б) $O(E\log V)$

В) $O((V + E)\log V)$

Г) $O(V \cdot E)$

3. У якому випадку BFS є ефективнішим для знаходження найкоротших шляхів, ніж алгоритм Дейкстри?

   А) У зважених графах з невід'ємними вагами.

   Б) У зважених графах з від'ємними вагами.

   В) У незважених графах.

   Г) У щільних графах.

4. Що означає, якщо `distance[U] + w_uv < distance[V]` під час релаксації ребра (U, V)?

   А) Вершина V вже була відвідана.

   Б) Знайдено коротший шлях до V через U.

   В) Ребро (U, V) є мостом.

   Г) Вершина U недосяжна.

5. Яка інформація зберігається в масиві **`predecessor`** після виконання алгоритму Дейкстри?

   А) Вага найкоротшого шляху до кожної вершини.

   Б) Час відкриття кожної вершини.

   В) Попередня вершина на найкоротшому шляху до поточної вершини.

   Г) Кількість ребер у найкоротшому шляху.

6. Яке з цих застосувань є типовим для алгоритму Дейкстри?

   А) Виявлення циклів у соціальних мережах.

   Б) Планування маршрутів у GPS-навігаторах.

   В) Знаходження сильно зв'язних компонент.

   Г) Топологічне сортування завдань.

7. Якщо в чергу з пріоритетом додається кілька записів для однієї й тієї ж вершини (з різними відстанями), як алгоритм Дейкстри обробляє це?

   А) Він обробляє всі записи для цієї вершини.

   Б) Він обробляє лише перший запис, а інші ігнорує.

   В) Він використовує перевірку `d > distance[U]` для ігнорування застарілих (довших) записів.

   Г) Це неможлива ситуація.

8. Який алгоритм слід використовувати для знаходження найкоротших шляхів у графі, який може містити від'ємні ваги ребер (але без від'ємних циклів)?

   А) BFS. Б) Алгоритм Флойда-Воршелла.

   В) Алгоритм Беллмана-Форда.

   Г) Алгоритм Прима.

9. Чому алгоритм Дейкстри є "жадібним"?

   А) Він завжди обирає ребро з найменшою вагою.

   Б) Він завжди обирає найближчу необроблену вершину, вважаючи, що це оптимальний вибір на даному етапі.

   В) Він намагається відвідати якомога більше вершин за один крок.

   Г) Він не виконує відкат.

10. Яка з цих проблем НЕ може бути вирішена безпосередньо алгоритмом Дейкстри?

    А) Знаходження найкоротшого шляху між двома містами на карті.

    Б) Визначення найшвидшого маршруту для пакетів даних у мережі.

    В) Знаходження найкоротшого шляху в графі з циклом від'ємної ваги.

    Г) Визначення найдешевшого маршруту доставки товарів з невід'ємною вартістю.

Відповіді:

1 Б). Його жадібний вибір може “фіналізувати” шлях до вершини, який пізніше виявиться неоптимальним через від’ємний цикл.

2 В). $O((V + E)\log V)$.

3 В). У незважених графах.

4 Б). Знайдено коротший шлях до V через U.

5 В). Попередня вершина на найкоротшому шляху до поточної вершини.

6 Б). Планування маршрутів у GPS-навігаторах.

7 В). Він використовує перевірку `d > distance[U]` для ігнорування застарілих (довших) записів.

8 В). Алгоритм Беллмана-Форда.

9 Б). Він завжди обирає найближчу необроблену вершину, вважаючи, що це оптимальний вибір на даному етапі.

10 В). Знаходження найкоротшого шляху в графі з циклом від’ємної ваги.

Завдання для самостійної роботи

1. Реалізація черги з пріоритетом: Реалізуй власну чергу з пріоритетом на основі бінарної купи. Інтегруй її в алгоритм Дейкстри.
2. Відновлення всіх шляхів: Модифікуй функцію `reconstructPath` так, щоб вона могла виводити всі найкоротші шляхи до заданої цільової вершини, якщо їх існує кілька з однаковою мінімальною вагою.
3. Візуалізація алгоритму: Створи простий інструмент для візуалізації роботи алгоритму Дейкстри: показуй граф, підсвічуй оброблювану вершину, оновлюй відстані та кольори ребер/вершин.
4. Дейкстра для всіх пар: Досліди, як алгоритм Дейкстри можна застосувати для знаходження найкоротших шляхів між усіма парами вершин у графі. Порівняй ефективність такого підходу з алгоритмом Флойда-Воршелла.

5. Оптимізація для щільних графів: Досліди, як алгоритм Дейкстри може бути оптимізований для щільних графів (де $E \approx V^2$) за допомогою використання масиву замість черги з пріоритетом для пошуку мінімальної відстані (або за допомогою фібоначчієвої купи).
6. Дейкстра на сітці: Застосуй алгоритм Дейкстри для пошуку найкоротшого шляху на сітці (наприклад, лабіринті), де кожна клітинка є вершиною, а перехід між клітинками має вагу.
7. Реалізація Алгоритму Беллмана-Форда: Реалізуй алгоритм Беллмана-Форда, який може працювати з від'ємними вагами ребер та виявляти цикли від'ємної ваги. Порівняй його з Дейкстрою.
8. Обмеження на кількість ребер: Модифікуй алгоритм Дейкстри, щоб він знаходив найкоротший шлях, але з обмеженням на максимальну кількість ребер у шляху.
9. Двонаправлений Дейкстра: Досліди та реалізуй двонаправлений алгоритм Дейкстри, який починає пошук одночасно з початкової та кінцевої вершин. Як це може прискорити пошук?
10. Реалізація на реальних даних: Знайди невеликий набір реальних географічних даних (наприклад, перехрестя та дороги в невеликому районі) і застосуй до них алгоритм Дейкстри для розрахунку маршрутів.

Алгоритм Беллмана-Форда для знаходження найкоротшого шляху

Після знайомства з алгоритмом Дейкстри, який ефективно знаходить найкоротші шляхи у графах з невід'ємними вагами ребер, ми переходимо до більш універсального, але й складнішого алгоритму — алгоритму Беллмана-Форда. Цей алгоритм є ключовим для розв'язання проблеми найкоротших шляхів у випадках, коли граф може містити ребра з від'ємними вагами.

1.1. Що таке Алгоритм Беллмана-Форда?

Алгоритм Беллмана-Форда — це алгоритм для знаходження найкоротших шляхів від однієї початкової вершини до всіх інших вершин у зваженому орієнтованому графі. Він був розроблений незалежно Річардом Беллманом

у 1958 році та Лестером Фордом-молодшим у 1956 році. Його ключова перевага над алгоритмом Дейкстри полягає в тому, що він може коректно працювати з графами, які містять ребра з від'ємними вагами.

1.2. Проблема найкоротшого шляху з від'ємними Вагами

Як ми вже знаємо, зважений граф — це граф, де кожному ребру присвоєно числове значення, його вага. Вага шляху — це сума ваг усіх ребер, що входять до цього шляху. Найкоротший шлях — це шлях з мінімальною загальною вагою.

Проблема виникає, коли граф містить ребра з від'ємною вагою. У таких графах алгоритм Дейкстри може дати некоректні результати, оскільки його жадібний підхід (фіналізація найкоротшого шляху до вершини) не враховує можливість того, що пізніше знайдений шлях через від'ємне ребро може скоротити вже "фіналізовану" відстань.

Ще складнішою ситуацією є наявність циклів від'ємної ваги. Цикл від'ємної ваги — це цикл, сума ваг ребер якого є від'ємним числом. Якщо такий цикл існує і досяжний з початкової вершини, то найкоротший шлях до будь-якої вершини, що належить цьому циклу (або досяжна з нього), стає мінус нескінченністю ($-\infty$). Це відбувається тому, що ви можете нескінченно багато разів проходити по цьому циклу, зменшуючи загальну вагу шляху до нескінченно малих значень. У таких випадках поняття "найкоротшого шляху" стає невизначеним. Алгоритм Беллмана-Форда здатний не тільки знаходити найкоротші шляхи за наявності від'ємних ваг ребер (без від'ємних циклів), а й виявляти наявність циклів від'ємної ваги.

1.3. Ітераційна релаксація алгоритму Беллмана-Форда

На відміну від жадібного підходу Дейкстри, Беллман-Форд використовує метод ітераційної релаксації. Він не "фіналізує" відстані до вершин на кожному кроці. Замість цього, він багаторазово проходить по всіх ребрах графа, виконуючи операцію релаксації для кожного ребра.

Якщо найкоротший шлях від початкової вершини S до будь-якої іншої вершини V містить k ребер, то цей шлях не може бути довшим, ніж $V - 1$ ребер (де V — загальна кількість вершин у графі), оскільки шлях без циклів може відвідати кожну вершину лише один раз.

Алгоритм Беллмана-Форда виконує $V - 1$ ітерацій. На кожній ітерації він переглядає всі ребра графа і намагається виконати операцію релаксації для кожного з них. Після k-ї ітерації, алгоритм гарантує, що він знайшов

найкоротші шляхи до всіх вершин, які можна досягти з початкової вершини за не більше ніж k ребер. Таким чином, після $V-1$ ітерацій (що є максимальною можливою кількістю ребер у простому шляху без циклів), алгоритм гарантує, що він знайшов найкоротші шляхи до всіх досяжних вершин, якщо в графі немає циклів від'ємної ваги.

Релаксація ребра (U, V) з вагою $w(U, V)$:

- Припустимо, у нас є поточна оцінка найкоротшого шляху від початкової вершини S до U, позначена як `distance[U]`.
- Якщо `distance[U]` не є нескінченністю (тобто, U досяжна), і `distance[U] + w(U, V) < distance[V]`, то ми оновлюємо `distance[V]` на `distance[U] + w(U, V)`.
- Ми також запам'ятовуємо, що шлях до V тепер проходить через U (зберігаючи `predecessor[V] = U`).

1.4. Виявлення циклів від'ємної ваги

Після виконання $V-1$ ітерацій, якщо ми можемо виконати ще хоча б одну релаксацію (тобто, на V-й ітерації знайдемо ребро (U, V), для якого `distance[U] + w(U, V) < distance[V]`), це означає, що в графі існує цикл від'ємної ваги, і цей цикл досяжний з початкової вершини. У такому випадку найкоротші шляхи не визначені.

1.5. Покроковий опис алгоритму Беллмана-Форда

1. Ініціалізація:
    - Створити масив `distance` (або `dist`) розміром V, де `distance[i]` зберігатиме поточну оцінку найкоротшого шляху від початкової вершини до вершини i.
    - Ініціалізувати `distance[start_node] = 0` (шлях до самої себе нульовий).
    - Ініціалізувати `distance[i] = НЕСКІНЧЕННІСТЬ` для всіх інших вершин i (оскільки шляхи до них ще невідомі).
    - Створити масив `predecessor` (або `prev`) розміром V, де `predecessor[i]` зберігатиме попередню вершину на найкоротшому шляху до i. Ініціалізувати всі як `NULL` або `-1`.
2. Основний цикл релаксації (V-1 ітерацій):

- Повторити `V-1` разів (де V — загальна кількість вершин у графі):
    - Для кожного ребра (U, V) з вагою w_{uv} у графі (перебрати всі ребра):
        - Якщо `distance[U]` не є `НЕСКІНЧЕННІСТЬ` (тобто, U досяжна з `start_node`):
            - Якщо `distance[U] + w_uv < distance[V]`:
                - `distance[V] = distance[U] + w_uv`
                - `predecessor[V] = U`

3. Перевірка на цикли від'ємної ваги (V-та ітерація):
    - Після завершення $V - 1$ ітерацій, виконати ще одну, V-ту ітерацію.
    - Для кожного ребра (U, V) з вагою w_{uv} у графі:
        - Якщо `distance[U]` не є `НЕСКІНЧЕННІСТЬ` (тобто, U досяжна з `start_node`):
            - Якщо `distance[U] + w_uv < distance[V]`:
                - Це означає, що ми знайшли коротший шлях, що вказує на існування циклу від'ємної ваги, досяжного з `start_node`.
                - Повернути повідомлення про наявність циклу від'ємної ваги (або спеціальне значення). Найкоротші шляхи не визначені.
4. Результат:
    - Якщо цикл від'ємної ваги не виявлено, `distance` масив містить найкоротші відстані від `start_node` до всіх досяжних вершин, а `predecessor` масив дозволяє відновити ці шляхи.
    - Повернути `distance`, `predecessor`.

1.6. Псевдокод Алгоритму Беллмана-Форда

Структури даних, що використовуються:

- `Graph G`: Вхідний зважений орієнтований граф.
- `G.Edges`: Список всіх ребер графа, де кожне ребро представлене трійкою `(U, V, w_uv)` (джерело, призначення, вага).
- `start_node`: Початкова вершина.
- `distance`: Масив розміром `V`, `distance[i]` зберігає поточну мінімальну відстань від `start_node` до `i`.
- `predecessor`: Масив розміром `V`, `predecessor[i]` зберігає попередню вершину на найкоротшому шляху до `i`.

```
// Допоміжні операції:

// Операція: КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)
// Призначення: Повертає загальну кількість вершин у Графі.

// Операція: ОТРИМАТИ_ВСІ_РЕБРА(Граф)
// Призначення: Повертає список всіх ребер графа. Кожне ребро предста
влене як (джерело, призначення, вага).

// Операція: ІНІЦІАЛІЗУВАТИ_МАСИВ(Масив, Значення)
// Призначення: Встановлює всі елементи Масиву на задане Значення.

// Константа для нескінченності (дуже велике число)
КОНСТАНТА НЕСКІНЧЕННІСТЬ = 999999999

---

АЛГОРИТМ BELLMAN_FORD(Graph G, start_node):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ МАСИВ distance РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  3. СТВОРИТИ МАСИВ predecessor РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  4. ВСІ_РЕБРА = ОТРИМАТИ_ВСІ_РЕБРА(G) // Отримуємо список всіх ребер
графа

  5. ІНІЦІАЛІЗУВАТИ_МАСИВ(distance, НЕСКІНЧЕННІСТЬ) // Всі відстані с
початку нескінченні
  6. ІНІЦІАЛІЗУВАТИ_МАСИВ(predecessor, NULL) // Попередників немає

  7. distance[start_node] = 0 // Відстань до початкової вершини - 0

  8. // Крок 1: Основний цикл релаксації (V-1 ітерацій)
  9. ДЛЯ і ВІД 1 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Повторюємо V-1 разів
```

```
      a. ДЛЯ КОЖНОГО РЕБРА (U, V, w_uv) З ВСІ_РЕБРА: // Перебираємо всі ребра графа
          i. ЯКЩО distance[U] != НЕСКІНЧЕННІСТЬ: // Якщо вершина U досяжна
              1. ЯКЩО distance[U] + w_uv < distance[V]: // Якщо знайдено коротший шлях до V через U
                  a. distance[V] = distance[U] + w_uv // Оновити відстань до V
                  b. predecessor[V] = U // Встановити U як попередника V

  10. // Крок 2: Перевірка на цикли від'ємної ваги (V-та ітерація)
  11. ДЛЯ КОЖНОГО РЕБРА (U, V, w_uv) З ВСІ_РЕБРА: // Перебираємо всі ребра графа ще раз
      a. ЯКЩО distance[U] != НЕСКІНЧЕННІСТЬ: // Якщо вершина U досяжна
          i. ЯКЩО distance[U] + w_uv < distance[V]: // Якщо можна ще релаксувати, то є цикл від'ємної ваги
              1. ПОВЕРНУТИ "Граф містить цикл від'ємної ваги, досяжний з початкової вершини."
              2. // Можна також позначити distance[V] = -НЕСКІНЧЕННІСТЬ для всіх вершин у циклі та досяжних з нього

  12. ПОВЕРНУТИ distance, predecessor // Повертаємо масиви відстаней та попередників
```

Операції та їх пояснення:

- **КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)**: Повертає загальну кількість вершин у графі G. Це значення використовується для визначення кількості ітерацій.
- **ОТРИМАТИ_ВСІ_РЕБРА(G)**: Ця операція повертає список усіх ребер графа. Для алгоритму Беллмана-Форда важливо мати можливість ефективно перебирати *всі* ребра на кожній ітерації. Якщо граф представлений списками суміжності, це може вимагати ітерації по всіх списках.
- **ІНІЦІАЛІЗУВАТИ_МАСИВ(Масив, Значення)**: Допоміжна операція для заповнення масивів **distance** та **predecessor** початковими значеннями.
- **НЕСКІНЧЕННІСТЬ**: Символічна константа, що представляє дуже велике число. Використовується для ініціалізації відстаней до невідомих вершин.

- `distance[start_node] = 0`: Відстань від початкової вершини до самої себе завжди нульова.
- `ДЛЯ і ВІД 1 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1)`: Це зовнішній цикл, який виконує `V-1` ітерацій. Кожна ітерація гарантує, що найкоротші шляхи, що складаються з не більше ніж `і` ребер, вже знайдені.
- `ДЛЯ КОЖНОГО РЕБРА (U, V, w_uv) З ВСІ_РЕБРА`: Це внутрішній цикл, який на кожній ітерації перебирає *всі* ребра графа.
- `ЯКЩО distance[U] != НЕСКІНЧЕННІСТЬ`: Ця перевірка гарантує, що ми намагаємося релаксувати ребро (U, V) лише якщо вершина U вже досяжна з початкової вершини. Якщо U недосяжна, то і V через U також не буде досяжна найкоротшим шляхом.
- `ЯКЩО distance[U] + w_uv < distance[V]`: Це умова релаксації. Вона перевіряє, чи шлях до V через U (`distance[U] + w_uv`) є коротшим за поточну відому найкоротшу відстань до V (`distance[V]`).
- `distance[V] = distance[U] + w_uv`: Якщо знайдено коротший шлях, ми оновлюємо `distance[V]` новим, меншим значенням.
- `predecessor[V] = U`: Ми запам’ятовуємо, що для досягнення V найкоротшим шляхом, ми прийшли з вершини U. Це дозволяє нам відновити сам шлях.
- `Крок 2: Перевірка на цикли від'ємної ваги`: Після $V - 1$ ітерацій, якщо алгоритм все ще може знайти ребро, яке можна релаксувати (тобто, `distance[U] + w_uv < distance[V]` все ще істинно для деякого ребра), це означає, що існує цикл від’ємної ваги, який досяжний з `start_node`. У цьому випадку найкоротші шляхи не визначені, і алгоритм повертає відповідне повідомлення.

1.7. Часова та Просторова Складність

Часова складність: Алгоритм Беллмана-Форда виконує $V - 1$ ітерацій, і на кожній ітерації він переглядає всі E ребер графа. Таким чином, загальна часова складність становить $O(V \cdot E)$.

- Для розріджених графів (де $E \approx V$), це приблизно $O(V^2)$.

– Для щільних графів (де $E \approx V^2$), це приблизно $O(V^3)$. Ця складність є вищою, ніж у алгоритму Дейкстри ($O((V + E)\log V)$), але Беллман-Форд має перевагу в обробці від'ємних ваг.

Просторова складність: Просторова складність алгоритму Беллмана-Форда становить $O(V + E)$ для зберігання графа (списків ребер або списків суміжності) та масивів `distance` і `predecessor`.

3. Область застосування

Алгоритм Беллмана-Форда є незамінним у ситуаціях, де ваги ребер можуть бути від'ємними, і необхідно знайти найкоротші шляхи, або виявити цикли від'ємної ваги.

1. Мережева маршрутизація:
    - Протоколи маршрутизації: Хоча алгоритм Дейкстри використовується в OSPF, алгоритм Беллмана-Форда (або його варіації, такі як RIP - Routing Information Protocol) використовується в деяких протоколах маршрутизації, де "вартість" маршруту може бути від'ємною (наприклад, якщо певний маршрут надає "бонус" або "знижку").
    - Виявлення петель маршрутизації: Цикли від'ємної ваги можуть вказувати на проблеми в конфігурації мережі, де пакети можуть нескінченно циркулювати.
2. Арбітраж у фінансових ринках:
    - Обмін валют: Якщо ви моделюєте обмін валют як граф, де вершини — це валюти, а ребра — курси обміну. Ваги ребер можуть бути логарифмами курсів обміну. Якщо ви можете перетворити одну валюту на іншу, а потім назад, і отримати прибуток, це відповідає циклу від'ємної ваги (сума логарифмів буде від'ємною). Алгоритм Беллмана-Форда може виявити такі можливості арбітражу.
3. Виявлення від'ємних циклів:
    - Аналіз залежностей: У системах планування або моделювання, де від'ємні ваги можуть представляти "вигоди" або "скорочення часу". Цикли від'ємної ваги можуть вказувати на нескінченні можливості оптимізації або логічні суперечності.

- Ігрова теорія: У деяких ігрових моделях, де гравці можуть отримувати "від'ємні" винагороди.

4. Аналіз задач з обмеженнями:

    - Системи різницевих обмежень: Де змінні пов'язані нерівностями вигляду $x_i - x_j \leq b_{ij}$. Цю проблему можна перетворити на задачу найкоротшого шляху в графі з від'ємними вагами, де цикли від'ємної ваги вказують на несумісні обмеження.

5. Приклади застосування

- Арбітраж на валютному ринку: Уявіть, що у вас є три валюти: USD, EUR, GBP. Ви можете обміняти USD на EUR, потім EUR на GBP, а потім GBP назад на USD. Якщо курси обміну такі, що після цієї послідовності операцій ви отримуєте більше USD, ніж мали спочатку, це є можливістю для арбітражу. Ми можемо змоделювати це як граф: вершини — валюти, ребра — можливі обміни. Вага ребра (U, V) може бути $-\log(\text{курс обміну } U \to V)$. Якщо ви виявите цикл, сума ваг якого від'ємна, це означає, що $\sum\log(\text{курс}) < 0$, або $\log(\prod\text{курс}) < 0$, що означає $\prod\text{курс} < 1$. Це вказує на цикл від'ємної ваги, який Беллман-Форд може виявити, сигналізуючи про можливість арбітражу.

– Виявлення проблем у мережевій маршрутизації (RIP): Протокол RIP (Routing Information Protocol) є одним з протоколів маршрутизації, який використовує алгоритм, подібний до Беллмана-Форда (вектор відстані). У цьому протоколі маршрутизатори обмінюються інформацією про відстані до інших мереж. Якщо в мережі виникає цикл від'ємної ваги (наприклад, через неправильну конфігурацію або збій), це може призвести до того, що пакети даних будуть нескінченно циркулювати в цьому циклі, ніколи не досягаючи призначення, або ж маршрутизатор буде постійно оновлювати свою таблицю маршрутизації, вважаючи, що він знайшов "нескінченно" короткий шлях. Алгоритм Беллмана-Форда, будучи основою таких протоколів, може виявити такі "петлі маршрутизації" через свою здатність знаходити цикли від'ємної ваги.

– Планування проектів з бонусами: У деяких проектах виконання певних завдань може не тільки вимагати ресурсів (позитивна вага), але й надавати "бонуси" або "скорочення часу" для інших завдань

(від'ємна вага). Наприклад, інвестиція в нове обладнання (позитивна вага) може значно прискорити подальші етапи (від'ємна вага). Якщо ви змоделюєте це як граф залежностей, де ваги ребер відображають зміни у вартості/часі, то цикл від'ємної ваги може вказувати на можливість нескінченного "покращення" або на логічну суперечність у моделі проекту, яку потрібно виявити та усунути.

Переваги та Недоліки

Переваги алгоритму Беллмана-Форда:

- Обробка від'ємних ваг ребер: Це його головна перевага. Алгоритм коректно знаходить найкоротші шляхи навіть за наявності ребер з від'ємними вагами, якщо немає циклів від'ємної ваги.
- Виявлення циклів від'ємної ваги: Він здатний виявити, чи існує цикл від'ємної ваги, досяжний з початкової вершини. Це є важливою діагностичною функцією.
- Простота реалізації: Хоча він менш інтуїтивний, ніж Дейкстра, його реалізація є досить прямолінійною, оскільки вона включає лише прості цикли по ребрах. Не потрібна складна структура даних, як черга з пріоритетом.

Недоліки алгоритму Беллмана-Форда:

- Повільніший за Дейкстру: Його часова складність $O(V \cdot E)$ є значно вищою, ніж $O((V + E)\log V)$ у Дейкстри (для бінарної купи). Це робить його менш придатним для дуже великих графів, особливо якщо ваги ребер завжди невід'ємні.
- Неефективний для щільних графів: Для щільних графів, де $E \approx V^2$, складність наближається до $O(V^3)$, що є дуже повільним.
- Не може знайти шляхи за наявності циклів від'ємної ваги: Якщо такий цикл існує і досяжний, алгоритм лише повідомляє про його наявність, але не може знайти "найкоротші" шляхи, оскільки вони прямують до $-\infty$.

Контрольні запитання

1. Основна перевага: Яка головна перевага алгоритму Беллмана-Форда над алгоритмом Дейкстри?

2. Обмеження на ваги: Чи може алгоритм Беллмана-Форда працювати з від'ємними вагами ребер? А з циклами від'ємної ваги?
3. Цикл від'ємної ваги: Що таке цикл від'ємної ваги, і чому його наявність унеможливлює визначення найкоротшого шляху?
4. Ітерації: Скільки ітерацій релаксації виконує алгоритм Беллмана-Форда в основному циклі, і чому саме стільки?
5. Виявлення циклу від'ємної ваги: Як алгоритм Беллмана-Форда виявляє наявність циклу від'ємної ваги?
6. Релаксація: Опишіть процес релаксації ребра (U, V) з вагою w_{uv} в контексті алгоритму Беллмана-Форда.
7. Ініціалізація: Як ініціалізуються масиви `distance` та `predecessor` на початку алгоритму Беллмана-Форда?
8. Часова складність: Яка часова складність алгоритму Беллмана-Форда? Від чого вона залежить?
9. Порівняння з Дейкстрою: Чому алгоритм Дейкстри є кращим вибором, якщо всі ваги ребер невід'ємні?
10. Застосування: Наведіть один приклад реальної задачі, де алгоритм Беллмана-Форда є необхідним.

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота №X: Реалізація Алгоритму Беллмана-Форда та Виявлення Циклів Від'ємної Ваги

2. Мета

Метою даної лабораторної роботи є набуття практичних навичок у реалізації алгоритму Беллмана-Форда для знаходження найкоротших шляхів у зважених орієнтованих графах, які можуть містити від'ємні ваги ребер, а також у виявленні циклів від'ємної ваги.

3. Завдання

а) Реалізація представлення зваженого орієнтованого графа: Створити клас або структуру для представлення орієнтованого зваженого

графа. Граф повинен зберігати список всіх ребер (джерело, призначення, вага) для легкого перебору.

b) Реалізація алгоритму Беллмана-Форда: Розробити функцію `bellmanFord(Graph G, int start_node)`, яка повертає масиви найкоротших відстаней та попередників, або повідомлення про наявність циклу від'ємної ваги.

c) Відновлення шляху: Реалізувати функцію `reconstructPath(int[] predecessor, int start_node, int target_node)`, яка відновлює та виводить найкоротший шлях.

d) Тестування: Протестувати реалізований алгоритм на різних зважених орієнтованих графах, включаючи випадки, де:

- Граф містить від'ємні ваги ребер, але без циклів від'ємної ваги.
- Граф містить цикл від'ємної ваги, досяжний з початкової вершини.
- Граф містить цикл від'ємної ваги, недосяжний з початкової вершини.
- Граф є незв'язним (деякі вершини недосяжні).

4. Опис

Вам необхідно розробити програмний додаток на обраній мові програмування. Додаток повинен дозволяти:

– Створювати орієнтовані зважені графи з заданою кількістю вершин та ребер.

– Додавати ребра з вагами (включаючи від'ємні) до графа.

– Викликати функцію `bellmanFord` для знаходження найкоротших шляхів від заданої початкової вершини.

– Виводити знайдені найкоротші відстані до всіх вершин.

– Виводити відновлений найкоротший шлях до заданої цільової вершини.

– Чітко повідомляти, якщо виявлено цикл від'ємної ваги.

Використання списків ребер або списків суміжності (з можливістю ефективного перебору всіх ребер) для представлення графа є обов'язковим.

5. Порядок Виконання

a) Підготовка:

   a. Створіть новий проект або файл для лабораторної роботи.

   b. Визначте, як будете представляти зважені орієнтовані графи. Рекомендується зберігати список всіх ребер у вигляді трійок `(джерело, призначення, вага)`.

b) Реалізація алгоритму Беллмана-Форда:

   a. Створіть функцію `bellmanFord(Graph G, int start_node)`.

   b. Ініціалізуйте `distance` масив (з `НЕСКІНЧЕННІСТЬ`) та `predecessor` масив (з `NULL`).

   c. Встановіть `distance[start_node] = 0`.

   d. Реалізуйте основний цикл `V-1` ітерацій, де на кожній ітерації перебираються всі ребра та виконується релаксація.

   e. Реалізуйте додаткову V-ту ітерацію для перевірки на цикли від'ємної ваги. Якщо цикл виявлено, функція повинна повернути спеціальне значення або кинути виняток.

c) Реалізація відновлення шляху:

   a. Створіть функцію `reconstructPath(int[] predecessor, int start_node, int target_node)`.

   b. Ця функція повинна рухатися назад від `target_node` до `start_node` за допомогою масиву `predecessor` і зібрати шлях. Вона також повинна обробляти випадки, коли шлях не існує (наприклад, `target_node` недосяжна або є частиною циклу від'ємної ваги).

d) Тестування:

- Створіть кілька екземплярів `Graph` для різних тестових сценаріїв:

        - Граф з позитивними вагами (для порівняння з Дейкстрою).
        - Граф з від’ємними вагами, але без циклів від’ємної ваги.
        - Граф з циклом від’ємної ваги, досяжним з `start_node`.
        - Граф з циклом від’ємної ваги, недосяжним з `start_node`.
    - Для кожного графа:
        - Викличте `bellmanFord` з різними `start_node`.
        - Виведіть масив `distance`.
        - Якщо немає циклу від’ємної ваги, викличте `reconstructPath` для кількох `target_node` і виведіть відновлені шляхи.
        - Чітко виведіть повідомлення про виявлення циклу від’ємної ваги, якщо він є.

6. Результат

Після виконання всіх завдань програми повинні виводити:

– Для кожного тестового графа та початкової вершини:

- Або список найкоротших відстаней від початкової вершини до всіх інших вершин (наприклад, `Відстані від 0: {0: 0, 1: -2, 2: 5, 3: INF}`).
- Або повідомлення про наявність циклу від’ємної ваги (наприклад, `Граф містить цикл від'ємної ваги, найкоротші шляхи не визначені.`).
- Якщо шляхи визначені, для кожної заданої цільової вершини: відновлений найкоротший шлях (наприклад, `Шлях до 2: 0 -> 1 -> 2`).

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:

    - Реалізовано структуру даних для представлення орієнтованого зваженого графа, що дозволяє ефективно перебирати всі ребра.

- Надано функціональну реалізацію алгоритму Беллмана-Форда, включаючи логіку виявлення циклів від'ємної ваги.
- Надано функцію для відновлення найкоротшого шляху.
- Код має бути добре структурованим, читабельним та містити необхідні коментарі, що пояснюють ключові частини реалізації.

2. Продемонструвати роботу додатку:
    - Продемонструвати роботу алгоритму Беллмана-Форда на 3-4 прикладах графів, що ілюструють різні випадки:
        - Граф з позитивними вагами (для порівняння).
        - Граф з від'ємними вагами, але без циклів від'ємної ваги.
        - Граф з циклом від'ємної ваги (досяжним з початкової вершини).
        - Граф з циклом від'ємної ваги (недосяжним з початкової вершини).
    - Показати коректність виводу найкоротших відстаней та відновлених шляхів, або чітке повідомлення про наявність циклу від'ємної ваги.
3. Бути готовим відповісти на питання:
    - Принцип роботи: Детально пояснити покрокову роботу алгоритму Беллмана-Форда, включаючи ініціалізацію, ітераційну релаксацію та перевірку на цикли від'ємної ваги.
    - Обробка від'ємних ваг: Пояснити, чому Беллман-Форд може працювати з від'ємними вагами ребер, на відміну від Дейкстри.
    - Виявлення циклів: Детально пояснити, як саме Беллман-Форд виявляє цикли від'ємної ваги.
    - Складність: Обґрунтувати часову та просторову складність алгоритму Беллмана-Форда.
    - Порівняння: Порівняти алгоритми Дейкстри та Беллмана-Форда, вказавши їхні переваги, недоліки та сфери застосування.

- Застосування: Навести 2-3 приклади реальних задач, де алгоритм Беллмана-Форда є ключовим, особливо у контексті фінансового арбітражу або мережевих протоколів.

Тести

1. Яке головне обмеження алгоритму Дейкстри, яке усуває алгоритм Беллмана-Форда?

   А) Не працює з орієнтованими графами.

   Б) Не працює з незв'язними графами.

   В) Не працює з від'ємними вагами ребер.

   Г) Не може знайти всі найкоротші шляхи.

2. Скільки ітерацій релаксації виконує алгоритм Беллмана-Форда в основному циклі?

   А) V

   Б) $V - 1$

   В) E

   Г) $V + E$

3. Що відбувається, якщо алгоритм Беллмана-Форда виявляє цикл від'ємної ваги, досяжний з початкової вершини?

   А) Він знаходить найкоротший шлях до всіх вершин.

   Б) Він переходить до наступної ітерації.

   В) Він повідомляє, що найкоротші шляхи не визначені.

   Г) Він автоматично видаляє цикл від'ємної ваги.

4. Яка часова складність алгоритму Беллмана-Форда?

   А) $O(V\log V)$

   Б) $O(E\log V)$

   В) $O(V \cdot E)$

   Г) $O(V + E)$

5. Для чого використовується V-та ітерація в алгоритмі Беллмана-Форда?

   А) Для фіналізації всіх відстаней.

   Б) Для перевірки наявності циклів від'ємної ваги.

   В) Для відновлення шляхів.

   Г) Для обчислення початкових відстаней.

Відповіді:1 - В), 2 - Б), 3 - В), 4 - В), 5 - Б).

Тести підвищеної складності

1. Чому алгоритм Беллмана-Форда гарантує знаходження найкоротших шляхів після $V - 1$ ітерацій (за відсутності негативних циклів)?

   А) Тому що на кожній ітерації він фіналізує відстань до однієї вершини.

   Б) Тому що найкоротший шлях без циклів може містити максимум $V - 1$ ребер.

   В) Тому що він використовує чергу з пріоритетом.

   Г) Тому що він перевіряє всі можливі шляхи.

2. Який з цих сценаріїв є типовим застосуванням алгоритму Беллмана-Форда?

   А) Планування маршруту в GPS-навігаторі без заторів.

   Б) Виявлення можливостей арбітражу на валютному ринку.

   В) Знаходження мінімального остовного дерева.

   Г) Пошук найкоротшого шляху в незваженому графі.

3. Якщо вершина U недосяжна з початкової вершини, чи буде релаксуватися ребро (U, V) в алгоритмі Беллмана-Форда?

   А) Так, завжди.

   Б) Ні, оскільки `distance[U]` буде `НЕСКІНЧЕННІСТЬ`.

   В) Тільки якщо V також недосяжна.

Г) Тільки на останній ітерації.

4. Який алгоритм є більш ефективним для знаходження найкоротших шляхів у розріджених графах з невід'ємними вагами ребер?

   А) Алгоритм Беллмана-Форда.

   Б) Алгоритм Дейкстри з бінарною купою.

   В) Алгоритм Флойда-Воршелла.

   Г) Алгоритм Крускала.

5. Що станеться, якщо алгоритм Беллмана-Форда виявить цикл від'ємної ваги, але цей цикл не досяжний з початкової вершини?

   А) Алгоритм все одно повідомить про цикл.

   Б) Алгоритм знайде найкоротші шляхи до всіх вершин коректно.

   В) Алгоритм зациклиться.

   Г) Алгоритм перейде до наступної початкової вершини.

6. Яка з цих властивостей НЕ є перевагою алгоритму Беллмана-Форда?

   А) Здатність виявляти цикли від'ємної ваги.

   Б) Простота реалізації порівняно з Дейкстрою.

   В) Коректна робота з від'ємними вагами ребер.

   Г) Висока швидкість виконання для великих графів.

7. У чому полягає основна відмінність між ітераційною релаксацією Беллмана-Форда та жадібним вибором Дейкстри?

   А) Беллман-Форд релаксує ребра лише один раз, Дейкстра - багато разів.

   Б) Беллман-Форд багаторазово релаксує *всі* ребра, тоді як Дейкстра релаксує ребра лише з вершини з мінімальною поточною відстанню.

   В) Беллман-Форд використовує чергу з пріоритетом, Дейкстра - звичайну чергу.

Г) Беллман-Форд працює з неорієнтованими графами, Дейкстра - з орієнтованими.

8. Якщо граф має від'ємні ваги ребер, але не має циклів від'ємної ваги, чи гарантує Беллман-Форд знаходження найкоротших шляхів?

   А) Ні, він може зациклитися.

   Б) Так, гарантує.

   В) Тільки для дуже малих графів.

   Г) Тільки якщо граф є повним.

9. Яка з цих структур даних є найбільш підходящою для зберігання графа для алгоритму Беллмана-Форда, якщо він має ефективно перебирати всі ребра?

   А) Матриця суміжності.

   Б) Список суміжності.

   В) Список всіх ребер.

   Г) Дерево.

10. Яка з цих проблем може бути перетворена на задачу найкоротшого шляху з від'ємними вагами, де цикли від'ємної ваги вказують на несумісні обмеження?

    А) Задача комівояжера.

    Б) Задача мінімального остовного дерева.

    В) Система різницевих обмежень.

    Г) Топологічне сортування.

Відповіді:

1 Б). Тому що найкоротший шлях без циклів може містити максимум $V - 1$ ребер.

2 Б). Виявлення можливостей арбітражу на валютному ринку.

3 Б). Ні, оскільки `distance[U]` буде `НЕСКІНЧЕННІСТЬ`.

4 Б). Алгоритм Дейкстри з бінарною купою.

5 Б). Алгоритм знайде найкоротші шляхи до всіх вершин коректно.

6 Г). Висока швидкість виконання для великих графів.

7 Б). Беллман-Форд багаторазово релаксує *всі* ребра, тоді як Дейкстра релаксує ребра лише з вершини з мінімальною поточною відстанню.

8 Б). Так, гарантує.

9 В). Список всіх ребер.

10 В). Система різницевих обмежень.

Завдання для самостійної роботи

1. Реалізація відновлення шляху для негативних циклів: Модифікуй функцію `reconstructPath` так, щоб у випадку виявлення циклу від'ємної ваги, вона могла вивести сам цикл (або хоча б одну вершину, що належить до нього).

2. Оптимізація для розріджених графів: Досліди та реалізуй оптимізацію Беллмана-Форда, яка відома як "Shortest Path Faster Algorithm" (SPFA). Вона використовує чергу для зберігання вершин, відстані до яких були оновлені, і може бути швидшою на розріджених графах, хоча в найгіршому випадку все ще $O(VE)$.

3. Беллман-Форд для всіх пар: Досліди, як алгоритм Беллмана-Форда можна застосувати для знаходження найкоротших шляхів між усіма парами вершин у графі (шляхом запуску його з кожної вершини). Порівняй ефективність такого підходу з алгоритмом Флойда-Воршелла.

4. Виявлення всіх вершин у негативному циклі: Якщо Беллман-Форд виявив негативний цикл, спробуй розширити функціонал, щоб знайти та вивести всі вершини, які належать до цього циклу.

5. Системи різницевих обмежень: Візьми набір нерівностей вигляду $x_i - x_j \leq b_{ij}$ і перетвори їх у граф. Застосуй алгоритм Беллмана-Форда, щоб знайти розв'язок (значення x_i) або виявити несумісність (цикл від'ємної ваги).

6. Граф обміну валют: Створи граф, де вершини — це валюти (наприклад, USD, EUR, GBP, JPY), а ребра — можливі обміни з певними курсами. Перетвори курси в логарифмічні ваги та застосуй Беллман-Форд для пошуку можливостей арбітражу.

7. Візуалізація релаксації: Створи простий інструмент для візуалізації роботи алгоритму Беллмана-Форда: показуй граф, підсвічуй поточне ребро, яке релаксується, та оновлюй значення відстаней біля вершин.

8. Обмеження на кількість ребер: Модифікуй алгоритм Беллмана-Форда, щоб він знаходив найкоротший шлях, але з обмеженням на максимальну кількість ребер у шляху (це природно вписується в його ітераційну структуру).

9. Порівняння з DFS/BFS: Для графів без ваг (або з однаковими вагами), порівняй результати Беллмана-Форда з BFS. Поясни, чому BFS є значно швидшим.

10. Застосування у мережевих протоколах: Досліди, як алгоритм Беллмана-Форда (або його варіації) використовується в реальних протоколах маршрутизації, таких як RIP. Зосередься на тому, як він обробляє зміни в мережі та виявляє петлі.

Алгоритм SPFA (Shortest Path Faster Algorithm)

Після знайомства з алгоритмом Дейкстри, який ефективно знаходить найкоротші шляхи у графах з невід'ємними вагами ребер, та алгоритмом Беллмана-Форда, що справляється з від'ємними вагами, ми переходимо до його оптимізованої версії — алгоритму SPFA (Shortest Path Faster Algorithm). Цей алгоритм є спробою покращити практичну продуктивність Беллмана-Форда, зберігаючи при цьому його здатність обробляти від'ємні ваги ребер та виявляти цикли від'ємної ваги.

1.1. Що таке Алгоритм SPFA?

Алгоритм SPFA (Shortest Path Faster Algorithm) — це алгоритм для знаходження найкоротших шляхів від однієї початкової вершини до всіх інших вершин у зваженому орієнтованому графі. Він є покращеною версією алгоритму Беллмана-Форда. Подібно до Беллмана-Форда, SPFA може коректно працювати з графами, які містять ребра з від'ємними вагами, і також здатний виявляти цикли від'ємної ваги.

Основна ідея SPFA полягає в тому, щоб уникнути багаторазового перегляду всіх ребер на кожній ітерації, як це робить Беллман-Форд. Замість цього, SPFA використовує чергу, щоб обробляти лише ті вершини, відстані до яких були нещодавно оновлені.

1.2. Оптимізація над Беллманом-Фордом

Нагадаємо, що алгоритм Беллмана-Форда виконує $V - 1$ ітерацій, і на кожній ітерації він переглядає *всі* E ребер графа. Це призводить до часової складності $O(V \cdot E)$. Однак, на практиці, на багатьох графах, більшість релаксацій відбуваються на ранніх ітераціях, а подальші ітерації Беллмана-Форда часто виконують багато зайвих перевірок ребер, які не призводять до оновлень відстаней.

SPFA оптимізує цей процес, використовуючи чергу (Queue). Вершина додається до черги лише тоді, коли її відстань до початкової вершини була оновлена (релаксована). Це означає, що ми релаксуємо ребра лише з тих вершин, чиї відстані потенційно могли змінитися, а отже, могли б призвести до подальших оновлень відстаней їхніх сусідів.

1.3. Релаксація з використанням черги

SPFA, як і Беллман-Форд, ґрунтується на операції релаксації ребра.

– Релаксація ребра (U, V) з вагою $w(U, V)$: Якщо `distance[U] + w(U, V) < distance[V]`, то ми оновлюємо `distance[V]` на `distance[U] + w(U, V)` та встановлюємо `predecessor[V] = U`.

Основний механізм SPFA:

1. Ініціалізація:
    - Створити масив `distance` (або `dist`) розміром V, ініціалізувати `distance[start_node] = 0` та `distance[i] = НЕСКІНЧЕННІСТЬ` для всіх інших вершин.
    - Створити масив `predecessor` (або `prev`) розміром V, ініціалізувати всі як `NULL` або `-1`.
    - Створити чергу (Queue) `Q`. Додати до `Q` початкову вершину `start_node`.
    - Створити булевий масив `in_queue` розміром V, ініціалізувати всі як `false`. `in_queue[i]` буде `true`, якщо вершина `i` наразі

перебуває в черзі. Встановити `in_queue[start_node] = true`.

- Створити масив `visit_count` (або `count`) розміром V, ініціалізувати всі як 0. `visit_count[i]` буде відстежувати, скільки разів вершина `i` була додана до черги. Це використовується для виявлення циклів від'ємної ваги. Встановити `visit_count[start_node] = 1`.

2. Основний цикл:
    - Поки `Q` не порожня:
        - Вилучити вершину U з початку черги `Q`.
        - Встановити `in_queue[U] = false` (оскільки U більше не в черзі).
3. Релаксація сусідів:
    - Для кожної сусідньої вершини V з U (тобто, для кожного ребра (U, V) з вагою $w(U, V)$):
        - Якщо `distance[U] + w(U, V) < distance[V]` (знайдено коротший шлях до V через U):
            - Оновити `distance[V] = distance[U] + w(U, V)`.
            - Оновити `predecessor[V] = U`.
            - Якщо `V` не перебуває в черзі (`in_queue[V] == false`):
                - Додати `V` до кінця черги `Q`.
                - Встановити `in_queue[V] = true`.
                - Збільшити `visit_count[V]` на 1.
                - Якщо `visit_count[V] >= V` (або `V-1` залежно від строгості):
                    - Це означає, що вершина V була додана до черги V або більше разів, що вказує на

існування циклу від'ємної ваги, досяжного з початкової вершини.

- Повернути повідомлення про наявність циклу від'ємної ваги.

4. Завершення:
    - Якщо цикл від'ємної ваги не виявлено, `distance` масив містить найкоротші відстані від `start_node` до всіх досяжних вершин, а `predecessor` масив дозволяє відновити ці шляхи.
    - Повернути `distance`, `predecessor`.

1.4. Псевдокод алгоритму SPFA

Структури даних, що використовуються:

- `Graph G`: Вхідний зважений орієнтований граф.
- `AdjacencyList G.Adj`: Список суміжності графа, де кожне ребро зберігає вагу (наприклад, `G.Adj[U]` містить пари `(V, w(U,V))`).
- `start_node`: Початкова вершина.
- `distance`: Масив розміром `V`, `distance[i]` зберігає поточну мінімальну відстань від `start_node` до `i`.
- `predecessor`: Масив розміром `V`, `predecessor[i]` зберігає попередню вершину на найкоротшому шляху до `i`.
- `Queue Q`: Звичайна черга (FIFO).
- `in_queue`: Булевий масив розміром `V`, `in_queue[i]` є `true`, якщо вершина `i` наразі перебуває в черзі.
- `visit_count`: Масив розміром `V`, `visit_count[i]` відстежує, скільки разів вершина `i` була додана до черги.

```
// Допоміжні операції:

// Операція: КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)
// Призначення: Повертає загальну кількість вершин у Графі.

// Операція: ІНІЦІАЛІЗУВАТИ_МАСИВ(Масив, Значення)
// Призначення: Встановлює всі елементи Масиву на задане Значення.
```

```
// Операція: ДОДАТИ_В_ЧЕРГУ(Черга, Елемент)
// Призначення: Додає Елемент до кінця Черги.

// Операція: ВИЛУЧИТИ_З_ПОЧАТКУ_ЧЕРГИ(Черга)
// Призначення: Вилучає та повертає елемент з початку Черги.

// Операція: ЧЕРГА_НЕ_ПОРОЖНЯ(Черга)
// Призначення: Перевіряє, чи Черга містить елементи.
// Повертає: true, якщо Черга не порожня, false інакше.

// Операція: ОТРИМАТИ_СУСІДІВ_ТА_ВАГИ(Graph G, Вершина U)
// Призначення: Повертає список пар (сусідня_вершина, вага_ребра) для
всіх вихідних ребер з Вершини U.
// Деталізація: Для списків суміжності - G.Adj[U] містить такі пари.

// Константа для нескінченності (дуже велике число)
КОНСТАНТА НЕСКІНЧЕННІСТЬ = 999999999



АЛГОРИТМ SPFA(Graph G, start_node):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ МАСИВ distance РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  3. СТВОРИТИ МАСИВ predecessor РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  4. СТВОРИТИ Queue Q
  5. СТВОРИТИ БУЛЕВИЙ МАСИВ in_queue РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  6. СТВОРИТИ МАСИВ visit_count РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН

  7. ІНІЦІАЛІЗУВАТИ_МАСИВ(distance, НЕСКІНЧЕННІСТЬ)
  8. ІНІЦІАЛІЗУВАТИ_МАСИВ(predecessor, NULL)
  9. ІНІЦІАЛІЗУВАТИ_МАСИВ(in_queue, false)
  10. ІНІЦІАЛІЗУВАТИ_МАСИВ(visit_count, 0)

  11. distance[start_node] = 0
  12. ДОДАТИ_В_ЧЕРГУ(Q, start_node)
  13. in_queue[start_node] = true
  14. visit_count[start_node] = 1

  15. ПОКИ ЧЕРГА_НЕ_ПОРОЖНЯ(Q):
      a. U = ВИЛУЧИТИ_З_ПОЧАТКУ_ЧЕРГИ(Q)
      b. in_queue[U] = false // U більше не в черзі

      c. // Релаксація сусідів U
      d. ДЛЯ КОЖНОЇ ПАРИ (V, w_uv) З ОТРИМАТИ_СУСІДІВ_ТА_ВАГИ(G, U):
```

```
// Для кожного ребра (U, V) з вагою w_uv
          i. ЯКЩО distance[U] != НЕСКІНЧЕННІСТЬ: // Якщо вершина U досяжна
              1. ЯКЩО distance[U] + w_uv < distance[V]: // Якщо знайдено коротший шлях до V через U
        a. distance[V] = distance[U] + w_uv // Оновити відстань до V
        b. predecessor[V] = U // Встановити U як попередника V

        c. ЯКЩО in_queue[V] == false: // Якщо V не перебуває в черзі
           i. ДОДАТИ_В_ЧЕРГУ(Q, V) // Додати V в чергу
           ii. in_queue[V] = true // Позначити V як таку, що перебуває в черзі
           iii. visit_count[V] = visit_count[V] + 1 // Збільшити лічильник відвідувань V

           iv. ЯКЩО visit_count[V] >= КІЛЬКІСТЬ_ВЕРШИН: // Якщо V додано до черги V або більше разів
  1. ПОВЕРНУТИ "Граф містить цикл від'ємної ваги, досяжний з початкової вершини."
  2. // Можна також позначити distance[V] = -НЕСКІНЧЕННІСТЬ для всіх вершин у циклі та досяжних з нього

  16. ПОВЕРНУТИ distance, predecessor // Повертаємо масиви відстаней та попередників
```

Операції та їх пояснення:

- `КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)`: Повертає загальну кількість вершин у графі G.
- `ІНІЦІАЛІЗУВАТИ_МАСИВ(Масив, Значення)`: Допоміжна операція для заповнення масивів початковими значеннями.
- `НЕСКІНЧЕННІСТЬ`: Символічна константа для початкових відстаней.
- `distance[start_node] = 0`: Відстань від початкової вершини до самої себе нульова.
- `Queue Q`: Звичайна черга (FIFO). Вона містить вершини, чиї відстані були нещодавно оновлені, і з яких, можливо, потрібно релаксувати ребра.
- `in_queue`: Булевий масив, який відстежує, чи перебуває вершина `i` наразі в черзі. Це запобігає додаванню однієї й тієї ж вершини до черги кілька разів, якщо вона вже там є.

- `visit_count`: Масив, який зберігає кількість разів, коли вершина була додана до черги. Це ключовий елемент для виявлення циклів від'ємної ваги. Якщо `visit_count[V]` стає рівним або більшим за `V` (загальну кількість вершин), це означає, що вершина V була релаксована надто багато разів, що можливо лише за наявності циклу від'ємної ваги, який проходить через V і досяжний з `start_node`.
- `ПОКИ ЧЕРГА_НЕ_ПОРОЖНЯ(Q)`: Основний цикл алгоритму. Він продовжується до тих пір, поки в черзі є вершини, з яких потрібно релаксувати ребра.
- `U = ВИЛУЧИТИ_З_ПОЧАТКУ_ЧЕРГИ(Q)`: Вилучаємо вершину U з черги для обробки.
- `in_queue[U] = false`: Після вилучення вершини з черги, її статус `in_queue` змінюється на `false`.
- `ЯКЩО distance[U] != НЕСКІНЧЕННІСТЬ`: Перевірка, чи U досяжна.
- `ЯКЩО distance[U] + w_uv < distance[V]`: Умова релаксації.
- `ЯКЩО in_queue[V] == false`: Ключова оптимізація SPFA. Вершина V додається до черги лише тоді, коли її відстань оновлена, І вона не перебуває в черзі. Якщо вона вже в черзі, це означає, що вона буде оброблена пізніше, і її відстань буде оновлена, коли її вилучать.
- `ЯКЩО visit_count[V] >= КІЛЬКІСТЬ_ВЕРШИН`: Умова виявлення циклу від'ємної ваги.

1.5. Часова та Просторова Складність

Часова складність:

- У найгіршому випадку: Часова складність SPFA залишається такою ж, як і у Беллмана-Форда: $O(V \cdot E)$. Існують спеціально сконструйовані графи, які можуть змусити SPFA працювати так само повільно.
- У середньому випадку (на практиці): На багатьох практичних графах (особливо розріджених), SPFA часто працює значно швидше, ніж Беллман-Форд, і його продуктивність може бути ближчою до $O(E)$ або $O(V + E)$. Однак, це не гарантовано.

Просторова складність: Просторова складність алгоритму SPFA становить $O(V + E)$ для зберігання графа (списків суміжності), масивів `distance`,

`predecessor`, `in_queue`, `visit_count`, а також черги `Q` (яка може містити до $O(V)$ елементів).

2. Область застосування

SPFA, як оптимізація Беллмана-Форда, застосовується в тих самих сценаріях, де потрібні найкоротші шляхи в графах з від'ємними вагами ребер, і де важлива вища практична продуктивність порівняно з Беллманом-Фордом.

1. Мережева маршрутизація:
    - Використовується в деяких протоколах маршрутизації, що базуються на векторних відстанях, де вартість маршруту може бути від'ємною або змінюватися динамічно.
    - Виявлення петель маршрутизації, спричинених від'ємними циклами.
2. Арбітраж у фінансових ринках:
    - Пошук можливостей арбітражу на валютних ринках або ринках цінних паперів, де комісії або курси обміну можуть призвести до від'ємних ваг (після логарифмування).
3. Виявлення негативних циклів:
    - У системах, де наявність циклу від'ємної ваги вказує на логічну суперечність, нескінченну оптимізацію або проблему (наприклад, у системах планування, моделюванні потоків).
4. Аналіз задач з обмеженнями:
    - Вирішення систем різницевих обмежень ($x_i - x_j \leq b_{ij}$), де цикли від'ємної ваги свідчать про несумісні обмеження.
5. Алгоритми потоків мінімальної вартості:
    - SPFA може бути використаний як підпрограма в деяких алгоритмах для знаходження потоків мінімальної вартості в мережах, де вартість може бути від'ємною.

3. Приклади застосування

– Оптимізація логістики з “бонусами”: Уявіть логістичну мережу, де переміщення товарів між деякими складами не тільки коштує грошей, але й може приносити “бонус” (наприклад, через податкові пільги, субсидії або утилізацію відходів, що перетворюється на прибуток). Цей “бонус” можна представити як від’ємну вагу ребра. Якщо існує послідовність переміщень, яка постійно приносить прибуток (цикл від’ємної ваги), це може бути виявлено SPFA. В іншому випадку, SPFA знайде найдешевший шлях, враховуючи ці бонуси.

– Виявлення аномалій у мережевих протоколах (наприклад, RIP): Хоча RIP є спрощеною версією, що базується на Беллмані-Форді, ідея подібна. Якщо в мережі маршрутизації через помилки конфігурації або збої виникають маршрути, які постійно зменшують “вартість” проходження (створюючи цикл від’ємної ваги), це може призвести до “чорних дірок” або нескінченних петель, де пакети даних застрягають. SPFA може бути використаний для моніторингу таких умов, швидко виявляючи, чи якась вершина (маршрутизатор) надто часто оновлює свою відстань, що свідчить про проблему.

– Аналіз виробничих процесів з оптимізацією ресурсів: На виробництві, де різні етапи (вершини) мають певні витрати (позитивні ваги), але деякі оптимізаційні кроки можуть “зменшувати” загальні витрати (від’ємні ваги). Якщо існує послідовність кроків, яка призводить до нескінченного зменшення витрат (цикл від’ємної ваги), це вказує на фундаментальну проблему в моделі процесу або на нескінченну можливість оптимізації, яку потрібно дослідити. SPFA може допомогти знайти найоптимальніший шлях виконання процесу або виявити такі аномалії.

Переваги та Недоліки

Переваги алгоритму SPFA:

– Обробка від’ємних ваг ребер: Як і Беллман-Форд, SPFA коректно знаходить найкоротші шляхи навіть за наявності ребер з від’ємними вагами, якщо немає циклів від’ємної ваги.

– Виявлення циклів від’ємної ваги: Він здатний ефективно виявляти, чи існує цикл від’ємної ваги, досяжний з початкової вершини.

– Практична ефективність: На багатьох практичних графах (особливо розріджених), SPFA працює значно швидше, ніж класичний Беллман-Форд, наближаючись до продуктивності Дейкстри.

– Простота реалізації: Його реалізація є відносно простою, оскільки вона базується на звичайній черзі, а не на складнішій черзі з пріоритетом.

Недоліки алгоритму SPFA:

– Найгірший випадок: У найгіршому випадку (на спеціально сконструйованих графах) його часова складність залишається $O(V \cdot E)$, що є значно повільніше, ніж Дейкстра для графів без від'ємних ваг.

– Не гарантована швидкість: На відміну від Дейкстри (з бінарною купою), яка має гарантовану продуктивність, SPFA не має таких гарантій у середньому випадку.

– Повільніший за Дейкстру на невід'ємних вагах: Якщо всі ваги ребер невід'ємні, алгоритм Дейкстри (з чергою з пріоритетом) завжди буде швидшим за SPFA.

– Не може знайти шляхи за наявності циклів від'ємної ваги: Якщо такий цикл існує і досяжний, алгоритм лише повідомляє про його наявність, але не може знайти "найкоротші" шляхи, оскільки вони прямують до $-\infty$.

Контрольні запитання

1. Відношення до Беллмана-Форда: Як алгоритм SPFA пов'язаний з алгоритмом Беллмана-Форда? Яку проблему він намагається вирішити?

2. Оптимізація: Яка ключова оптимізація використовується в SPFA порівняно з Беллманом-Фордом? Яка структура даних лежить в її основі?

3. Виявлення циклу від'ємної ваги: Як SPFA виявляє наявність циклу від'ємної ваги? Який масив для цього використовується?

4. Масив `in_queue`: Для чого використовується булевий масив `in_queue` в алгоритмі SPFA?

5. Часова складність: Яка часова складність алгоритму SPFA у найгіршому випадку? А яка його типова продуктивність на практиці?
6. Порівняння з Дейкстрою: Які переваги та недоліки SPFA порівняно з алгоритмом Дейкстри?
7. Релаксація: Опишіть, коли відбувається релаксація ребра (U, V) в алгоритмі SPFA.
8. Ініціалізація: Які масиви ініціалізуються на початку SPFA, і які значення їм присвоюються?
9. Застосування: Наведіть один приклад, де SPFA може бути кращим вибором, ніж Дейкстра.
10. Обмеження: Чи може SPFA знайти найкоротші шляхи, якщо граф містить цикл від'ємної ваги, досяжний з початкової вершини?

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота №X: Реалізація Алгоритму SPFA для Пошуку Найкоротших Шляхів та Виявлення Циклів Від'ємної Ваги

2. Мета

Метою даної лабораторної роботи є набуття практичних навичок у реалізації алгоритму SPFA, розуміння його оптимізацій порівняно з Беллманом-Фордом, а також його здатності знаходити найкоротші шляхи у графах з від'ємними вагами ребер та виявляти цикли від'ємної ваги.

3. Завдання

a) Реалізація представлення зваженого орієнтованого графа: Створити клас або структуру для представлення орієнтованого зваженого графа за допомогою списків суміжності, де кожне ребро має вагу.

b) Реалізація алгоритму SPFA: Розробити функцію **`spfa(Graph G, int start_node)`**, яка повертає масиви найкоротших відстаней та попередників, або повідомлення про наявність циклу від'ємної ваги.

c) Відновлення шляху: Реалізувати функцію **`reconstructPath(int[] predecessor, int start_node, int target_node)`**, яка відновлює та виводить найкоротший шлях.

d) Тестування: Протестувати реалізований алгоритм на різних зважених орієнтованих графах, включаючи випадки, де:

- Граф містить від'ємні ваги ребер, але без циклів від'ємної ваги.
- Граф містить цикл від'ємної ваги, досяжний з початкової вершини.
- Граф містить цикл від'ємної ваги, недосяжний з початкової вершини.
- Порівняти продуктивність SPFA з Беллманом-Фордом на розріджених графах.

4. Опис

Вам необхідно розробити програмний додаток на обраній мові програмування (наприклад, Python, C++, Java, JavaScript). Додаток повинен дозволяти:

– Створювати орієнтовані зважені графи з заданою кількістю вершин та ребер.

– Додавати ребра з вагами (включаючи від'ємні) до графа.

– Викликати функцію **`spfa`** для знаходження найкоротших шляхів від заданої початкової вершини.

– Виводити знайдені найкоротші відстані до всіх вершин.

– Виводити відновлений найкоротший шлях до заданої цільової вершини.

– Чітко повідомляти, якщо виявлено цикл від'ємної ваги.

Використання списків суміжності для представлення графа є обов'язковим.

5. Порядок Виконання

а) Підготовка:

- Створіть новий проект або файл для лабораторної роботи.
- Визначте, як будете представляти зважені орієнтовані графи (наприклад, **`Map<Integer, List<Pair<Integer, Integer>>>`** де **`Pair`** це (**`сусід, вага`**)).

b) Реалізація алгоритму SPFA:

- Створіть функцію `spfa(Graph G, int start_node)`.
- Ініціалізуйте `distance`, `predecessor`, `Q`, `in_queue`, `visit_count` масиви/структури.
- Встановіть `distance[start_node] = 0`, додайте `start_node` до `Q`, оновіть `in_queue` та `visit_count`.
- Реалізуйте основний цикл `ПОКИ Q НЕ ПОРОЖНЯ`, включаючи вилучення вершини, релаксацію сусідів, умовне додавання до черги та перевірку `visit_count` для виявлення циклів від'ємної ваги.

c)Реалізація відновлення шляху:

- Створіть функцію `reconstructPath(int[] predecessor, int start_node, int target_node)`.
- Ця функція повинна рухатися назад від `target_node` до `start_node` за допомогою масиву `predecessor` і зібрати шлях. Вона також повинна обробляти випадки, коли шлях не існує.

d) Тестування:

- Створіть кілька екземплярів `Graph` для різних тестових сценаріїв:
  - Граф з позитивними вагами (для порівняння з Дейкстрою).
  - Граф з від'ємними вагами, але без циклів від'ємної ваги.
  - Граф з циклом від'ємної ваги, досяжним з `start_node`.
  - Граф з циклом від'ємної ваги, недосяжним з `start_node`.
- Для кожного графа:
  - Викличте `spfa` з різними `start_node`.
  - Виведіть масив `distance`.

        - Якщо немає циклу від'ємної ваги, викличте `reconstructPath` для кількох `target_node` і виведіть відновлені шляхи.
        - Чітко виведіть повідомлення про виявлення циклу від'ємної ваги, якщо він є.
    - Порівняння: Запустіть SPFA та Беллман-Форд (якщо реалізовано в попередній лабі) на одному розрідженому графі з від'ємними вагами та порівняйте час виконання.

5. Результат

Після виконання всіх завдань програми повинні виводити:

– Для кожного тестового графа та початкової вершини:

- Або список найкоротших відстаней від початкової вершини до всіх інших вершин.
- Або повідомлення про наявність циклу від'ємної ваги.
- Якщо шляхи визначені, для кожної заданої цільової вершини: відновлений найкоротший шлях.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:
    - Реалізовано структуру даних для представлення орієнтованого зваженого графа за допомогою списків суміжності.
    - Надано функціональну реалізацію алгоритму SPFA, включаючи логіку виявлення циклів від'ємної ваги.
    - Надано функцію для відновлення найкоротшого шляху.
    - Код має бути добре структурованим, читабельним та містити необхідні коментарі.
2. Продемонструвати роботу додатку:
    - Продемонструвати роботу алгоритму SPFA на 3-4 прикладах графів, що ілюструють різні випадки: з від'ємними вагами без циклів, з циклами від'ємної ваги (досяжними та недосяжними).

- Показати коректність виводу найкоротших відстаней та відновлених шляхів, або чітке повідомлення про наявність циклу від'ємної ваги.
- Якщо реалізовано Беллман-Форд: Продемонструвати порівняння часу виконання SPFA та Беллмана-Форда на одному розрідженому графі з від'ємними вагами, щоб показати оптимізацію SPFA.

3. Бути готовим відповісти на питання:
    - Принцип роботи: Детально пояснити покрокову роботу алгоритму SPFA, включаючи ініціалізацію, використання черги, масиву `in_queue` та `visit_count`, а також процес релаксації.
    - Оптимізація: Пояснити, як SPFA оптимізує роботу Беллмана-Форда, та чому він часто швидший на практиці.
    - Виявлення циклів: Детально пояснити, як саме SPFA виявляє цикли від'ємної ваги.
    - Складність: Обґрунтувати часову та просторову складність алгоритму SPFA у найгіршому та середньому (практичному) випадку.
    - Порівняння: Порівняти SPFA з Дейкстрою та Беллманом-Фордом, вказавши їхні переваги, недоліки та сфери застосування.
    - Застосування: Навести 2-3 приклади реальних задач, де алгоритм SPFA є корисним.

Тести

1. Яка структура даних є ключовою для оптимізації алгоритму SPFA?

    А) Стек.

    Б) Черга з пріоритетом.

    В) Звичайна черга (FIFO).

    Г) Хеш-таблиця.

2. Яку проблему вирішує алгоритм SPFA?

А) Знаходження мінімального остовного дерева.

Б) Знаходження найкоротших шляхів у графах з невід'ємними вагами.

В) Знаходження найкоротших шляхів у графах з від'ємними вагами та виявлення циклів від'ємної ваги.

Г) Топологічне сортування.

3. Який масив використовується в SPFA для відстеження кількості разів, коли вершина була додана до черги, з метою виявлення циклів від'ємної ваги?

   А) `distance`

   Б) `predecessor`

   В) `in_queue`

   Г) `visit_count`

4. У найгіршому випадку часова складність алгоритму SPFA дорівнює:

   А) $O(V \log V)$

   Б) $O(E \log V)$

   В) $O(V \cdot E)$

   Г) $O(V + E)$

5. Коли вершина V додається до черги в алгоритмі SPFA?

   А) Коли її відстань оновлюється, і вона вже перебуває в черзі.

   Б) Коли її відстань оновлюється, і вона НЕ перебуває в черзі.

   В) Коли її відстань не оновлюється.

   Г) На кожній ітерації для всіх вершин.

Відповіді: 1 - В), 2 - В), 3 - Г), 4 - В), 5 - Б).

Тести підвищеної складності

1. Чому SPFA часто швидший за Беллмана-Форда на практиці, незважаючи на однакову найгіршу часову складність?

А) SPFA використовує багатопотоковість.

Б) SPFA уникає релаксації ребер з вершин, відстані до яких не змінилися.

В) SPFA використовує фібоначчієву купу.

Г) SPFA перевіряє лише частину ребер на кожній ітерації.

2. Яке твердження вірно щодо `visit_count[V]` в алгоритмі SPFA?

   А) Воно рахує кількість ребер у найкоротшому шляху до V.

   Б) Воно рахує кількість разів, коли вершина V була вилучена з черги.

   В) Воно рахує кількість разів, коли вершина V була додана до черги.

   Г) Воно вказує, чи V перебуває у циклі від'ємної ваги.

3. Якщо SPFA виявляє, що `visit_count[V] >= V` для деякої вершини V, що це означає?

   А) Знайдено найкоротший шлях до V.

   Б) Вершина V недосяжна.

   В) Граф містить цикл від'ємної ваги, досяжний з початкової вершини.

   Г) Алгоритм завершився успішно.

4. Який з цих алгоритмів є найкращим вибором для знаходження найкоротших шляхів у щільних графах з від'ємними вагами ребер?

   А) Алгоритм Дейкстри.

   Б) Алгоритм SPFA.

   В) Алгоритм Беллмана-Форда.

   Г) Алгоритм Флойда-Воршелла (для всіх пар).

5. У чому полягає основна відмінність між чергою в BFS та чергою в SPFA?

   А) Черга в BFS зберігає вершини, а в SPFA - пари (відстань, вершина).

Б) Черга в BFS використовується для обходу, а в SPFA - для релаксації.

В) Черга в BFS є чергою з пріоритетом, а в SPFA - звичайною чергою.

Г) Черга в BFS не дозволяє повторів, а в SPFA - дозволяє.

6. Яке з цих застосувань є типовим для SPFA?

   А) Пошук найкоротшого маршруту для автомобіля в GPS.

   Б) Виявлення арбітражних можливостей на валютному ринку.

   В) Побудова мінімального остовного дерева.

   Г) Топологічне сортування завдань проекту.

7. Якщо граф не містить циклів від'ємної ваги, чи гарантує SPFA знаходження найкоротших шляхів?

   А) Ні, він може зациклитися.

   Б) Так, гарантує.

   В) Тільки для дуже малих графів.

   Г) Тільки якщо граф є повним.

8. Який масив використовується в SPFA для відстеження, чи вершина вже перебуває в черзі, щоб уникнути її повторного додавання?

   А) `distance`

   Б) `predecessor`

   В) `in_queue`

   Г) `visit_count`

9. Чому SPFA не може знайти "найкоротші" шляхи, якщо існує цикл від'ємної ваги, досяжний з початкової вершини?

   А) Тому що алгоритм зациклюється.

   Б) Тому що шлях може бути нескінченно коротким (прямувати до $-\infty$).

   В) Тому що він не може обробляти від'ємні числа.

Г) Тому що він знаходить найдовший шлях у цьому випадку.

10. Яке твердження вірно щодо продуктивності SPFA на розріджених графах порівняно з Дейкстрою (з бінарною купою) для графів без від'ємних ваг?

А) SPFA завжди швидший.

Б) Дейкстра завжди швидший.

В) SPFA може бути швидшим у середньому, але Дейкстра має кращу гарантовану складність.

Г) Їхня продуктивність однакова.

Відповіді:

1 Б). SPFA уникає релаксації ребер з вершин, відстані до яких не змінилися.

2 В). Воно рахує кількість разів, коли вершина V була додана до черги.

3 В). Граф містить цикл від'ємної ваги, досяжний з початкової вершини.

4 В). Алгоритм Беллмана-Форда.

5 Б). Черга в BFS використовується для обходу, а в SPFA - для релаксації.

6 Б). Виявлення арбітражних можливостей на валютному ринку.

7 Б). Так, гарантує.

8 В). `in_queue`.

9 Б). Тому що шлях може бути нескінченно коротким (прямувати до $-\infty$).

10 Б). Дейкстра завжди швидший.

Завдання для самостійної роботи

1. Порівняння продуктивності: Реалізуй алгоритми Беллмана-Форда та SPFA. Створи кілька тестових графів (розріджені, щільні, з від'ємними вагами, з негативними циклами) і порівняй час виконання обох алгоритмів.

2. Візуалізація SPFA: Створи простий інструмент для візуалізації роботи алгоритму SPFA: показуй граф, підсвічуй вершину, яка вилучається з черги, оновлюй відстані та показуй стан черги.

3. Виявлення всіх вершин у негативному циклі: Модифікуй SPFA, щоб у випадку виявлення негативного циклу, він не просто повідомляв про його наявність, а й виводив усі вершини, які належать до цього циклу.

4. SPFA для всіх пар: Досліди, як алгоритм SPFA можна застосувати для знаходження найкоротших шляхів між усіма парами вершин у графі. Порівняй ефективність такого підходу з алгоритмом Флойда-Воршелла.

5. Оптимізації SPFA: Досліди та реалізуй одну з додаткових оптимізацій SPFA, наприклад, “Small-Label First” або “Large-Label Last”, які змінюють порядок вилучення вершин з черги, щоб покращити середню продуктивність.

6. Застосування в задачах потоків: Досліди, як SPFA може бути використаний як підпрограма в алгоритмах для знаходження потоків мінімальної вартості в мережах.

7. SPFA для систем різницевих обмежень: Візьми набір нерівностей вигляду $x_i - x_j \leq b_{ij}$ і перетвори їх у граф. Застосуй SPFA, щоб знайти розв’язок (значення x_i) або виявити несумісність (цикл від’ємної ваги).

8. Відновлення шляху для циклів від’ємної ваги: Якщо виявлено цикл від’ємної ваги, спробуй модифікувати функцію відновлення шляху, щоб вона могла показати цей цикл.

9. Проблема найдовшого шляху в DAG: Досліди, як SPFA (або його модифікація) може бути використаний для знаходження найдовшого шляху в орієнтованому ациклічному графі (DAG), перетворивши ваги ребер.

10. Реалізація на реальних даних: Знайди невеликий набір даних, що моделюють фінансові обміни або логістику з “бонусами”, і застосуй до них SPFA для розрахунку найоптимальніших шляхів або виявлення аномалій.

Найкоротші шляхи (всі пари)

Алгоритм Флойд-Воршелла

Тепер ми переходимо до потужного алгоритму, який здатний розв'язати проблему знаходження найкоротших шляхів не лише від однієї початкової вершини, а й між усіма парами вершин у графі. Це алгоритм Флойд-Воршелла.

Алгоритм Флойд-Воршелла — це алгоритм динамічного програмування, призначений для знаходження найкоротших шляхів між усіма парами вершин у зваженому орієнтованому графі.

Він був розроблений незалежно Робертом Флойдом у 1962 році та Стефаном Воршеллом у 1962 році. Цей алгоритм є одним із класичних рішень проблеми "всі пари найкоротших шляхів" (All-Pairs Shortest Path, APSP).

1.2. Проблема "Всі Пари Найкоротших Шляхів" (APSP)

У попередніх розділах ми зосередилися на проблемі "одного джерела найкоротших шляхів" (SSSP), яку вирішують алгоритми Дейкстри та Беллмана-Форда. Проблема APSP є більш загальною: ми хочемо знайти найкоротший шлях від кожної вершини до кожної іншої вершини в графі.

Ключові особливості проблеми APSP:

- Зважений граф: Ребра мають ваги (вартості, відстані, час).
- Орієнтований або неорієнтований: Алгоритм Флойд-Воршелла працює з орієнтованими графами. Для неорієнтованого графа кожне неорієнтоване ребро (U, V) з вагою w можна представити як два орієнтованих ребра (U, V) та (V, U) з вагою w.
- Ваги ребер: Алгоритм Флойд-Воршелла може коректно працювати з ребрами, що мають від'ємні ваги.
- Цикли від'ємної ваги: Подібно до Беллмана-Форда, алгоритм Флойд-Воршелла не може знайти найкоротші шляхи, якщо граф містить цикли від'ємної ваги. Однак він здатний виявити їхню наявність. Якщо такий цикл існує і досяжний, найкоротший шлях до/з вершин у цьому циклі стає мінус нескінченністю.

1.3. Алгоритм Флойд-Воршелла -динамічне програмування

Алгоритм Флойд-Воршелла базується на принципах динамічного програмування. Він будує рішення ітеративно, поступово покращуючи оцінки найкоротших шляхів.

Алгоритм розглядає вершини графа одну за одною як потенційні проміжні вершини на шляху між будь-якою парою інших вершин.

Нехай $dist[i][j]$ буде поточною оцінкою найкоротшого шляху від вершини i до вершини j. Алгоритм виконує V ітерацій. На k-й ітерації (де k — це індекс вершини, що розглядається як проміжна), алгоритм перевіряє, чи шлях від i до j, що проходить тільки через вершини з індексами від 0 до $k-1$ (або від 1 до k), може бути покращений шляхом використання вершини k як проміжної.

Тобто, для кожної пари вершин (i, j), алгоритм запитує: "Чи шлях від i до j, що проходить через вершину k, коротший, ніж поточний найкоротший шлях від i до j, який не використовує k (або використовує лише проміжні вершини з меншими індексами)?"

Формально, для кожної пари (i, j): $dist[i][j] = \min(dist[i][j], dist[i][k] + dist[k][j])$

Де:

- `dist[i][j]` — це поточна найкоротша відстань від i до j.
- `dist[i][k]` — це найкоротша відстань від i до k, використовуючи лише проміжні вершини з індексами, меншими за k.
- `dist[k][j]` — це найкоротша відстань від k до j, використовуючи лише проміжні вершини з індексами, меншими за k.

Після того, як зовнішній цикл по k завершиться для всіх вершин (від 0 до $V-1$), `dist[i][j]` буде містити довжину найкоротшого шляху від i до j, використовуючи будь-яку вершину як проміжну.

1.4. Виявлення циклів від'ємної ваги

Алгоритм Флойд-Воршелла також може виявляти цикли від'ємної ваги. Якщо після завершення всіх V ітерацій, для будь-якої вершини i, `dist[i][i]` стає від'ємним, це означає, що існує цикл від'ємної ваги, який включає вершину i. Це відбувається тому, що якщо ви можете пройти по циклу від'ємної ваги, ви можете нескінченно зменшувати вагу шляху, повертаючись до початкової вершини.

1.5. Покроковий опис алгоритму Флойд-Воршелла

1. Ініціалізація матриць:
    - Створити матрицю `dist[V][V]` (розміром $V \times V$), де `dist[i][j]` зберігатиме найкоротшу відстань від вершини i до вершини j.
    - Створити матрицю `predecessor[V][V]` (розміром $V \times V$), де `predecessor[i][j]` зберігатиме попередню вершину на найкоротшому шляху від i до j. Це потрібно для відновлення шляхів.
    - Заповнити матрицю `dist`:
        - Для кожного i від 0 до $V - 1$:
            - Для кожного j від 0 до $V - 1$:
                - Якщо $i == j$: `dist[i][j] = 0` (відстань до самої себе).
                - Якщо існує пряме ребро від i до j з вагою w_{ij}: `dist[i][j] = w_{ij}`.
                - Інакше: `dist[i][j] = НЕСКІНЧЕННІСТЬ` (якщо прямого ребра немає).
    - Заповнити матрицю `predecessor`:
        - Для кожного i від 0 до $V - 1$:
            - Для кожного j від 0 до $V - 1$:
                - Якщо $i == j$: `predecessor[i][j] = i` (або `NULL`, оскільки шлях до себе не має попередника).
                - Якщо існує пряме ребро від i до j: `predecessor[i][j] = i`.
                - Інакше: `predecessor[i][j] = NULL` (або `-1`, якщо шлях невідомий).
2. Основний цикл (V ітерацій):

- Для `k` від 0 до $V - 1$ (проміжна вершина):
    - Для `i` від 0 до $V - 1$ (початкова вершина):
        - Для `j` від 0 до $V - 1$ (кінцева вершина):
            - Якщо `dist[i][k]` не `НЕСКІНЧЕННІСТЬ` І `dist[k][j]` не `НЕСКІНЧЕННІСТЬ`:
                - Якщо `dist[i][k] + dist[k][j] < dist[i][j]`:
                    - `dist[i][j] = dist[i][k] + dist[k][j]`
                    - `predecessor[i][j] = predecessor[k][j]` (або `k`, залежить від реалізації відновлення шляху)

3. Перевірка на цикли від'ємної ваги:
    - Після завершення всіх ітерацій:
        - Для кожного i від 0 до $V - 1$:
            - Якщо `dist[i][i] < 0`:
                - Граф містить цикл від'ємної ваги. Найкоротші шляхи не визначені.
                - (Опціонально) Можна позначити всі `dist[i][j]` як $-\infty$ для вершин, що належать до циклу від'ємної ваги або досяжні з нього.
4. Результат:
    - Якщо цикл від'ємної ваги не виявлено, матриця `dist` містить найкоротші відстані між усіма парами вершин.
    - Матриця `predecessor` дозволяє відновити ці шляхи.
    - Повернути `dist`, `predecessor`.

1.6. Псевдокод алгоритму Флойд-Воршелла

Структури даних, що використовуються:

- `Graph G`: Вхідний зважений орієнтований граф.
- `V`: Загальна кількість вершин у графі.
- `dist[V][V]`: Матриця, що зберігає найкоротші відстані.
- `predecessor[V][V]`: Матриця, що зберігає попередників для відновлення шляхів.

```
// Допоміжні операції:

// Операція: КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)
// Призначення: Повертає загальну кількість вершин у Графі.

// Операція: ОТРИМАТИ_ВАГУ_РЕБРА(Граф, ДЖЕРЕЛО, ПРИЗНАЧЕННЯ)
// Призначення: Повертає вагу прямого ребра від ДЖЕРЕЛА до ПРИЗНАЧЕНН
Я.
// Повертає: Вагу ребра, або НЕСКІНЧЕННІСТЬ, якщо ребра немає.

// Константа для нескінченності (дуже велике число)
КОНСТАНТА НЕСКІНЧЕННІСТЬ = 999999999


АЛГОРИТМ FLOYD_WARSHALL(Graph G):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ МАТРИЦЮ dist РОЗМІРОМ [КІЛЬКІСТЬ_ВЕРШИН][КІЛЬКІСТЬ_ВЕРШ
ИН]
  3. СТВОРИТИ МАТРИЦЮ predecessor РОЗМІРОМ [КІЛЬКІСТЬ_ВЕРШИН][КІЛЬКІС
ТЬ_ВЕРШИН]

  4. // Крок 1: Ініціалізація матриць
  5. ДЛЯ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. ДЛЯ j ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
          i. ЯКЩО i == j:
              1. dist[i][j] = 0
              2. predecessor[i][j] = i // Або NULL, або спеціальне зн
ачення
          ii. ІНАКШЕ:
              1. w_ij = ОТРИМАТИ_ВАГУ_РЕБРА(G, i, j)
              2. dist[i][j] = w_ij
              3. ЯКЩО w_ij != НЕСКІНЧЕННІСТЬ:
                  a. predecessor[i][j] = i
              b. ІНАКШЕ:
                  a. predecessor[i][j] = NULL // Або -1, або спеціаль
```

```
не значення

  6. // Крок 2: Основний цикл (V ітерацій)
  7. ДЛЯ k ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Проміжна вершина
      a. ДЛЯ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Початкова вершина
          i. ДЛЯ j ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1): // Кінцева вершин
а
              1. ЯКЩО dist[i][k] != НЕСКІНЧЕННІСТЬ І dist[k][j] != НЕ
СКІНЧЕННІСТЬ: // Перевірка на досяжність
                  a. ЯКЩО dist[i][k] + dist[k][j] < dist[i][j]:
                      i. dist[i][j] = dist[i][k] + dist[k][j]
                      ii. predecessor[i][j] = predecessor[k][j] // Он
овлюємо попередника

  8. // Крок 3: Перевірка на цикли від'ємної ваги
  9. ДЛЯ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. ЯКЩО dist[i][i] < 0:
          i. ПОВЕРНУТИ "Граф містить цикл від'ємної ваги."
          ii. // Можна також позначити всі відповідні dist[i][j] як -
НЕСКІНЧЕННІСТЬ

  10. ПОВЕРНУТИ dist, predecessor // Повертаємо матриці відстаней та
попередників
```

Операції та їх пояснення:

- **КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)**: Повертає загальну кількість вершин V у графі G. Це необхідно для визначення розміру матриць.
- **ОТРИМАТИ_ВАГУ_РЕБРА(Граф, ДЖЕРЕЛО, ПРИЗНАЧЕННЯ)**: Ця функція повертає вагу прямого ребра від **ДЖЕРЕЛА** до **ПРИЗНАЧЕННЯ**. Якщо такого ребра немає, вона повертає **НЕСКІНЧЕННІСТЬ**. Це може бути реалізовано шляхом доступу до матриці суміжності або перевіркою списків суміжності.
- **НЕСКІНЧЕННІСТЬ**: Символічна константа, що представляє дуже велике число, яке більше за будь-яку можливу суму ваг шляху. Використовується для ініціалізації відстаней до невідомих шляхів.
- **dist[V][V]**: Це основна матриця, яка на кожному кроці **k** зберігає найкоротші відстані між парами вершин (i, j), використовуючи лише проміжні вершини з індексами, меншими за **k**. Після завершення всіх ітерацій, вона міститиме остаточні найкоротші відстані.

– `predecessor[V][V]`: Ця матриця використовується для відновлення фактичних найкоротших шляхів. `predecessor[i][j]` зберігає вершину, яка передує j на найкоротшому шляху від i до j.

– Ініціалізація `dist` та `predecessor`:

  - `dist[i][j] = 0` для $i = j$: Відстань від вершини до самої себе нульова.
  - `dist[i][j] = w_ij` для прямого ребра: Початкова відстань між безпосередньо з'єднаними вершинами — це вага ребра.
  - `dist[i][j] = НЕСКІНЧЕННІСТЬ` для відсутніх ребер: Початково, якщо немає прямого ребра, шлях вважається нескінченним.
  - `predecessor[i][j] = i` (якщо ребро існує): Початково, якщо є пряме ребро, попередником j на шляху від i є сама i.
  - `predecessor[i][j] = NULL` (якщо ребра немає): Якщо прямого ребра немає, попередник не визначений.

– Три вкладені цикли (`k`, `i`, `j`): Це серце алгоритму.

  - Зовнішній цикл по `k`: Цей цикл перебирає всі можливі вершини графа, роблячи їх потенційними проміжними вершинами. Важливо, що `k` має бути зовнішнім циклом, оскільки на k-й ітерації ми припускаємо, що всі найкоротші шляхи, які використовують проміжні вершини з індексами меншими за k, вже обчислені.
  - Внутрішні цикли по `i` та `j`: Ці цикли перебирають усі можливі пари початкової (`i`) та кінцевої (`j`) вершин.

– `ЯКЩО dist[i][k] != НЕСКІНЧЕННІСТЬ І dist[k][j] != НЕСКІНЧЕННІСТЬ`: Ця перевірка гарантує, що шлях $i \to k \to j$ є дійсним (тобто, i може досягти k, і k може досягти j).

– `ЯКЩО dist[i][k] + dist[k][j] < dist[i][j]`: Це основна умова релаксації. Вона перевіряє, чи шлях від i до j, що проходить через проміжну вершину k, є коротшим за поточний відомий найкоротший шлях від i до j.

– `dist[i][j] = dist[i][k] + dist[k][j]`: Якщо знайдено коротший шлях, відстань оновлюється.

– `predecessor[i][j] = predecessor[k][j]`: Якщо шлях через k виявився коротшим, то попередником j на шляху від i є той самий попередник, що й на шляху від k до j. Це дозволяє коректно відновити шлях.

– Перевірка на цикли від'ємної ваги (`dist[i][i] < 0`): Після всіх ітерацій, якщо діагональний елемент `dist[i][i]` стає від'ємним, це означає, що існує цикл від'ємної ваги, який проходить через вершину i. У цьому випадку найкоротші шляхи не визначені.

1.7. Часова та просторова складність

Часова складність: Алгоритм Флойд-Воршелла має три вкладені цикли, кожен з яких виконується V разів. Таким чином, загальна часова складність становить $O(V^3)$. Це робить його придатним для графів з невеликою кількістю вершин (зазвичай до кількох сотень).

Просторова складність: Алгоритм Флойд-Воршелла використовує дві матриці розміром $V \times V$ (`dist` та `predecessor`). Таким чином, просторова складність становить $O(V^2)$.

3. Область застосування

Алгоритм Флойд-Воршелла є універсальним рішенням для знаходження найкоротших шляхів між усіма парами вершин, особливо коли граф може містити від'ємні ваги ребер.

1. Маршрутизація в невеликих мережах:
    - У невеликих комп'ютерних мережах, де кожен вузол повинен знати найкоротший шлях до кожного іншого вузла.
    - У телекомунікаційних мережах для розрахунку шляхів.
2. Транзитивне замикання графа:
    - Алгоритм може бути легко адаптований для знаходження транзитивного замикання графа (чи існує шлях від i до j), замінивши операції додавання на логічне "І", а мінімуму на логічне "АБО".
3. Виявлення циклів від'ємної ваги:

- Як вже згадувалося, алгоритм ефективно виявляє наявність таких циклів.

4. Обчислювальна біологія:
    - У задачах вирівнювання послідовностей ДНК або білків, де ваги можуть бути від'ємними (наприклад, штрафи за невідповідності).
5. Управління проектами:
    - У деяких моделях, де ваги можуть відображати "бонуси" або "штрафи" за виконання завдань, і потрібно знати найкоротший шлях між будь-якими двома етапами.
6. Графові бази даних:
    - Для виконання запитів, що стосуються найкоротших шляхів між елементами.

7. Приклади застосування

– Міжміська навігація (для невеликої кількості міст): Уявіть, що ви розробляєте навігаційну систему для невеликої країни або регіону, де є обмежена кількість міст (наприклад, до 200-300). Вам потрібно швидко знаходити найкоротші маршрути між *будь-якою парою* міст. Міста — це вершини, дороги — ребра, а відстані або час у дорозі — ваги. Алгоритм Флойд-Воршелла може бути запущений один раз, щоб обчислити всі найкоротші шляхи, і потім ці результати зберігаються в пам'яті для швидкого доступу. Це дозволяє миттєво відповідати на запити користувачів щодо маршрутів.

– Аналіз соціальних мереж (для малих кластерів): У невеликому кластері соціальної мережі (наприклад, група друзів з 50-100 осіб), ви можете використовувати Флойд-Воршелла, щоб визначити "відстань" (наприклад, мінімальну кількість посередників) між будь-якими двома людьми. Якщо "відстань" може бути від'ємною (наприклад, якщо певні зв'язки "скорочують" соціальну дистанцію), Флойд-Воршелл впорається.

– Виявлення арбітражу (розширений): Подібно до Беллмана-Форда, Флойд-Воршелл може бути використаний для виявлення

можливостей арбітражу на фінансових ринках. Якщо ви моделюєте валюти як вершини, а курси обміну як ваги ребер (перетворені на логарифми, щоб сума стала від'ємною для прибутку), то наявність циклу від'ємної ваги, виявленого Флойд-Воршеллом, вказуватиме на можливість арбітражу. Перевага Флойд-Воршелла в тому, що він виявляє це для всіх можливих циклів, а не лише тих, що досяжні з однієї початкової точки.

Переваги та Недоліки

Переваги алгоритму Флойд-Воршелла:

- Знаходить всі пари найкоротших шляхів: Це його основне призначення.
- Обробляє від'ємні ваги ребер: Коректно працює з графами, що містять від'ємні ваги ребер, за умови відсутності циклів від'ємної ваги.
- Виявляє цикли від'ємної ваги: Ефективно ідентифікує наявність таких циклів.
- Простота реалізації: Його реалізація є відносно простою та елегантною, використовуючи лише три вкладені цикли.
- Компактний код: Зазвичай вимагає менше рядків коду порівняно з багаторазовим запуском SSSP алгоритмів.

Недоліки алгоритму Флойд-Воршелла:

- Висока часова складність: $O(V^3)$ робить його непридатним для дуже великих графів (більше кількох сотень вершин).
- Висока просторова складність: $O(V^2)$ для зберігання матриць відстаней та попередників. Це може бути проблемою для графів з великою кількістю вершин, навіть якщо вони розріджені.
- Неефективний для розріджених графів: Якщо граф дуже розріджений (мало ребер), запуск алгоритму Дейкстри V разів (для всіх вершин як джерела) може бути швидшим, ніж Флойд-Воршелл, якщо всі ваги невід'ємні. Запуск Беллмана-Форда V разів також може бути швидшим для розріджених графів з від'ємними вагами (хоча це $O(V^2E)$).

Контрольні Запитання

1. Призначення Флойд-Воршелла: Яку проблему вирішує алгоритм Флойд-Воршелла?
2. Типи ваг: Які типи ваг ребер може обробляти алгоритм Флойд-Воршелла?
3. Динамічне програмування: Поясніть, як алгоритм Флойд-Воршелла використовує принцип динамічного програмування. Яку роль відіграє змінна `k`?
4. Виявлення циклів від'ємної ваги: Як алгоритм Флойд-Воршелла виявляє наявність циклу від'ємної ваги?
5. Ініціалізація: Як ініціалізується матриця **`dist`** на початку алгоритму?
6. Часова складність: Яка часова складність алгоритму Флойд-Воршелла?
7. Просторова складність: Яка просторова складність алгоритму Флойд-Воршелла?
8. Порівняння з Дейкстрою: Чому для розріджених графів з невід'ємними вагами ребер може бути ефективніше запустити алгоритм Дейкстри V разів, ніж використовувати Флойд-Воршелла?
9. Відновлення шляху: Як можна відновити найкоротший шлях між двома вершинами за допомогою матриці **`predecessor`**?
10. Застосування: Наведіть один приклад реальної задачі, де алгоритм Флойд-Воршелла є корисним.

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота №X: Реалізація Алгоритму Флойд-Воршелла для Знаходження Найкоротших Шляхів Між Усіма Парами Вершин

2. Мета

Метою даної лабораторної роботи є набуття практичних навичок у реалізації алгоритму Флойд-Воршелла для знаходження найкоротших шляхів між усіма парами вершин у зважених орієнтованих графах, включаючи обробку від'ємних ваг ребер та виявлення циклів від'ємної ваги.

3. Завдання

a) Реалізація представлення зваженого орієнтованого графа: Створити клас або структуру для представлення орієнтованого зваженого графа за допомогою матриці суміжності (для зручності роботи з Флойд-Воршеллом).

b) Реалізація алгоритму Флойд-Воршелла: Розробити функцію `floydWarshall(Graph G)`, яка повертає матриці найкоротших відстаней та попередників, або повідомлення про наявність циклу від'ємної ваги.

c) Відновлення шляху: Реалізувати функцію `reconstructPath(int[][] predecessor, int start_node, int target_node)`, яка відновлює та виводить найкоротший шлях між заданою парою вершин.

d) Тестування: Протестувати реалізований алгоритм на різних зважених орієнтованих графах, включаючи випадки, де:

- Граф містить позитивні ваги.
- Граф містить від'ємні ваги ребер, але без циклів від'ємної ваги.
- Граф містить цикл від'ємної ваги.
- Граф є незв'язним.

4. Опис

Вам необхідно розробити програмний додаток на обраній мові програмування (наприклад, Python, C++, Java, JavaScript). Додаток повинен дозволяти:

– Створювати орієнтовані зважені графи з заданою кількістю вершин.

– Додавати ребра з вагами (включаючи від'ємні) до графа.

– Викликати функцію `floydWarshall` для обчислення всіх найкоротших шляхів.

– Виводити отриману матрицю найкоротших відстаней.

– Виводити відновлений найкоротший шлях між заданими парами вершин.

– Чітко повідомляти, якщо виявлено цикл від'ємної ваги.

Використання матриці суміжності для представлення графа є обов'язковим для цієї лабораторної роботи.

5. Порядок Виконання
6. Підготовка:
    - Створіть новий проект або файл для лабораторної роботи.
    - Визначте, як будете представляти зважені орієнтовані графи за допомогою матриці суміжності (наприклад, `int[][] adjacencyMatrix`).
7. Реалізація алгоритму Флойд-Воршелла:
    - Створіть функцію `floydWarshall(Graph G)`.
    - Ініціалізуйте матриці `dist` та `predecessor` відповідно до початкових ваг ребер.
    - Реалізуйте три вкладені цикли (`k`, `i`, `j`) для виконання основної логіки алгоритму.
    - Реалізуйте перевірку на цикли від'ємної ваги після завершення основних циклів.
8. Реалізація відновлення шляху:
    - Створіть функцію `reconstructPath(int[][] predecessor, int start_node, int target_node)`.
    - Ця функція повинна рухатися назад від `target_node` до `start_node` за допомогою матриці `predecessor`, щоб зібрати шлях. Вона повинна обробляти випадки, коли шлях не існує (наприклад, `target_node` недосяжна).
9. Тестування:
    - Створіть кілька екземплярів `Graph` для різних тестових сценаріїв:
        - Граф з позитивними вагами.
        - Граф з від'ємними вагами, але без циклів від'ємної ваги.

        - Граф з циклом від'ємної ваги.
        - Незв'язний граф.
    - Для кожного графа:
        - Викличте `floydWarshall`.
        - Виведіть матрицю `dist`.
        - Якщо немає циклу від'ємної ваги, викличте `reconstructPath` для кількох пар `(start_node, target_node)` і виведіть відновлені шляхи.
        - Чітко виведіть повідомлення про виявлення циклу від'ємної ваги, якщо він є.

10. Результат

Після виконання всіх завдань програми повинні виводити:

– Для кожного тестового графа:

- Або матрицю найкоротших відстаней (`dist`).
- Або повідомлення про наявність циклу від'ємної ваги.
- Якщо шляхи визначені, для кожної заданої пари `(start_node, target_node)`: відновлений найкоротший шлях.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:
    - Реалізовано структуру даних для представлення орієнтованого зваженого графа за допомогою матриці суміжності.
    - Надано функціональну реалізацію алгоритму Флойд-Воршелла, включаючи логіку виявлення циклів від'ємної ваги.
    - Надано функцію для відновлення найкоротшого шляху між будь-якою парою вершин.
    - Код має бути добре структурованим, читабельним та містити необхідні коментарі.
2. Продемонструвати роботу додатку:

- Продемонструвати роботу алгоритму Флойд-Воршелла на 3-4 прикладах графів, що ілюструють різні випадки: з позитивними вагами, з від'ємними вагами без циклів, з циклами від'ємної ваги, незв'язні графи.
- Показати коректність виводу матриці відстаней та відновлених шляхів, або чітке повідомлення про наявність циклу від'ємної ваги.

3. Бути готовим відповісти на питання:
    - Принцип роботи: Детально пояснити покрокову роботу алгоритму Флойд-Воршелла, включаючи ініціалізацію матриць, роль проміжної вершини k та принцип динамічного програмування.
    - Обробка від'ємних ваг: Пояснити, як Флойд-Воршелл справляється з від'ємними вагами ребер.
    - Виявлення циклів: Детально пояснити, як алгоритм Флойд-Воршелла виявляє цикли від'ємної ваги.
    - Складність: Обґрунтувати часову та просторову складність алгоритму Флойд-Воршелла.
    - Порівняння: Порівняти Флойд-Воршелл з Дейкстрою (запущеною V разів) та Беллманом-Фордом (запущеним V разів) з точки зору їхніх переваг, недоліків та сфер застосування (особливо для розріджених/щільних графів).
    - Застосування: Навести 2-3 приклади реальних задач, де алгоритм Флойд-Воршелла є корисним.

Тести

1. Яку проблему вирішує алгоритм Флойд-Воршелла?

    А) Найкоротший шлях від одного джерела до всіх інших.

    Б) Найкоротший шлях між усіма парами вершин.

    В) Мінімальне остовне дерево.

    Г) Топологічне сортування.

2. Яка часова складність алгоритму Флойд-Воршелла?

   А) $O(V^2)$

   Б) $O(E^2)$

   В) $O(V^3)$

   Г) $O(V \cdot E)$

3. Алгоритм Флойд-Воршелла може працювати з графами, що містять:

   А) Тільки позитивні ваги ребер.

   Б) Тільки від'ємні ваги ребер.

   В) Позитивні та від'ємні ваги ребер, але без циклів від'ємної ваги.

   Г) Цикли від'ємної ваги.

4. Яка структура даних зазвичай використовується для зберігання відстаней в алгоритмі Флойд-Воршелла?

   А) Список суміжності.

   Б) Черга з пріоритетом.

   В) Матриця.

   Г) Стек.

5. Якщо `dist[i][i]` стає від'ємним після виконання алгоритму Флойд-Воршелла, що це означає?

   А) Шлях від i до i є найкоротшим.

   Б) Граф містить цикл від'ємної ваги.

   В) Вершина i недосяжна.

   Г) Алгоритм завершився успішно.

Відповіді: 1 - Б), 2 - В), 3 - В), 4 - В), 5 - Б).

Тести підвищеної складності

1. Чому у трьох вкладених циклах алгоритму Флойд-Воршелла зовнішній цикл обов’язково має бути по змінній `k` (проміжній вершині)?

   А) Щоб забезпечити коректний порядок релаксації, гарантуючи, що шляхи через менші індекси вже оптимізовані.

   Б) Щоб зменшити просторову складність.

   В) Щоб прискорити виявлення циклів від’ємної ваги.

   Г) Це не має значення, порядок циклів може бути довільним.

2. Яка просторова складність алгоритму Флойд-Воршелла?

   А) $O(V)$

   Б) $O(E)$

   В) $O(V^2)$

   Г) $O(V + E)$

3. Яке твердження є вірним щодо порівняння Флойд-Воршелла та запуску Дейкстри V разів на розрідженому графі з невід’ємними вагами?

   А) Флойд-Воршелл завжди швидший.

   Б) Запуск Дейкстри V разів буде швидшим.

   В) Їхня продуктивність однакова.

   Г) Флойд-Воршелл не може працювати з невід’ємними вагами.

4. Яке застосування є типовим для алгоритму Флойд-Воршелла?

   А) Маршрутизація пакетів у великих комп’ютерних мережах.

   Б) Знаходження найкоротших шляхів у GPS-навігаторах (мільйони перехресть).

   В) Обчислення часу подорожі між усіма парами перехресть у невеликому місті.

Г) Визначення мінімального остовного дерева.

5. Що означає `predecessor[i][j] = predecessor[k][j]` при оновленні шляху через проміжну вершину `k`?

   А) Вершина `k` є безпосереднім попередником `j` на шляху від `i`.

   Б) Шлях від `i` до `j` тепер проходить через `k`, і попередник `j` на цьому новому шляху є тим самим, що й попередник `j` на шляху від `k`.

   В) Вершина `j` є попередником `k`.

   Г) Шлях від `i` до `j` не змінився.

6. Який з цих алгоритмів є найбільш підходящим для знаходження найкоротших шляхів між усіма парами вершин у графі, що містить цикли від'ємної ваги?

   А) Алгоритм Дейкстри.

   Б) Алгоритм Беллмана-Форда.

   В) Алгоритм Флойд-Воршелла.

   Г) Жоден з цих алгоритмів не може знайти найкоротші шляхи, але Флойд-Воршелл може виявити цикли.

7. Якщо граф є незв'язним, як Флойд-Воршелл відобразить це у матриці `dist`?

   А) Відстані до недосяжних вершин будуть 0.

   Б) Відстані до недосяжних вершин будуть від'ємними.

   В) Відстані до недосяжних вершин залишаться `НЕСКІНЧЕННІСТЬ`.

   Г) Алгоритм видасть помилку.

8. Який з цих алгоритмів є прикладом динамічного програмування?

   А) BFS.

   Б) DFS.

   В) Алгоритм Флойд-Воршелла.

   Г) Алгоритм Прима.

9. Яка з цих проблем може бути вирішена шляхом адаптації алгоритму Флойд-Воршелла?

   А) Пошук Ейлерового циклу.

   Б) Знаходження транзитивного замикання графа.

   В) Побудова мінімального остовного дерева.

   Г) Пошук Гамільтонового шляху.

10. Порівняно з алгоритмом Беллмана-Форда, запущеним V разів, Флойд-Воршелл має:

    А) Кращу часову складність для всіх типів графів.

    Б) Гіршу часову складність для розріджених графів.

    В) Гіршу часову складність для щільних графів.

    Г) Кращу просторову складність.

Відповіді:

1 А). Щоб забезпечити коректний порядок релаксації, гарантуючи, що шляхи через менші індекси вже оптимізовані.

2 В). $O(V^2)$.

3 Б). Запуск Дейкстри V разів буде швидшим.

4 В). Обчислення часу подорожі між усіма парами перехресть у невеликому місті.

5 Б). Шлях від `i` до `j` тепер проходить через `k`, і попередник `j` на цьому новому шляху є тим самим, що й попередник `j` на шляху від `k`.

6 Г). Жоден з цих алгоритмів не може знайти найкоротші шляхи, але Флойд-Воршелл може виявити цикли.

7 В). Відстані до недосяжних вершин залишаться `НЕСКІНЧЕННІСТЬ`.

8 В). Алгоритм Флойд-Воршелла.

9 Б). Знаходження транзитивного замикання графа.

10 Б). Гіршу часову складність для розріджених графів.

Завдання для самостійної роботи

1. Відновлення шляху: Реалізуй функцію `reconstructPath(int[][] predecessor, int start_node, int target_node)` так, щоб вона виводила повний шлях від `start_node` до `target_node`. Обробляй випадки, коли шлях не існує.

2. Транзитивне замикання: Модифікувати алгоритм Флойд-Воршелла, щоб він обчислював транзитивне замикання графа. Це означає, що для кожної пари (i, j) він повинен визначити, чи існує шлях від i до j.

3. Виявлення всіх негативних циклів: Розширити алгоритм Флойд-Воршелла, щоб він не просто повідомляв про наявність негативного циклу, а й міг ідентифікувати та виводити всі вершини, що належать до будь-якого циклу від'ємної ваги.

4. Центр графа: Використай результати Флойд-Воршелла, щоб знайти центр графа (вершину, яка мінімізує максимальну відстань до будь-якої іншої вершини) та діаметр графа (найдовший з усіх найкоротших шляхів).

5. Візуалізація: Створи простий інструмент для візуалізації роботи алгоритму Флойд-Воршелла. Показуй матрицю `dist` та `predecessor` на кожній ітерації `k`, підсвічуючи оновлені значення.

6. Оптимізація пам'яті (для дуже великих графів): Досліди, як можна зменшити просторову складність, якщо потрібно лише знайти найкоротші відстані, а не відновлювати шляхи (наприклад, не зберігати `predecessor` матрицю).

7. Порівняння продуктивності: Для графів з різною кількістю вершин (наприклад, 10, 50, 100, 200), порівняй час виконання Флойд-Воршелла з багатократним запуском алгоритму Дейкстри (для невід'ємних ваг) та Беллмана-Форда (для від'ємних ваг).

8. Матричне множення: Досліди, як алгоритм Флойд-Воршелла може бути реалізований за допомогою повторного матричного множення (min-plus матричне множення).

9. Задача про найдовший шлях у DAG: Модифікуй алгоритм Флойд-Воршелла, щоб він знаходив найдовші шляхи між усіма парами вершин в орієнтованому ациклічному графі (DAG).

10. Застосування у реальних даних: Знайди невеликий набір реальних даних (наприклад, дані про польоти між кількома аеропортами, або маршрути громадського транспорту) та застосуй алгоритм Флойд-Воршелла для розрахунку найкоротших шляхів між усіма парами точок.

Алгоритм Джонсона

Алгоритм Джонсона — це ефективний алгоритм для розв'язання проблеми "всі пари найкоротших шляхів" (All-Pairs Shortest Path, APSP) у зваженому орієнтованому графі. Він здатний працювати з графами, що містять ребра з від'ємними вагами, за умови, що в графі відсутні цикли від'ємної ваги. Якщо такі цикли присутні, алгоритм Джонсона може їх виявити.

Алгоритм Джонсона є комбінацією двох інших відомих алгоритмів: алгоритму Беллмана-Форда та алгоритму Дейкстри. Він використовує алгоритм Беллмана-Форда для "переважування" ребер графа таким чином, щоб усі ваги стали невід'ємними, а потім запускає алгоритм Дейкстри з кожної вершини на цьому переваженому графі.

Алгоритм Джонсона є більш ефективним ніж алгоритм Флойд-Воршелла для розріджених графів (де кількість ребер E значно менше, ніж V^2), оскільки його продуктивність для таких графів краща, ніж у Флойд-Воршелла.

1.1. Алгоритм Джонсона - переважування та багаторазовий Дейкстра

Алгоритм Джонсона складається з трьох основних кроків:

Крок 1: Додавання нової вершини та запуск Беллмана-Форда

1. Створення нового графа G': Створюється допоміжний граф G', який є копією оригінального графа G. До нього додається одна нова вершина s, яка не належить до оригінальних вершин графа.

2. Додавання нульових ребер: З нової вершини s додаються орієнтовані ребра до кожної оригінальної вершини $v \in V$ з вагою 0. Тобто, для кожного $v \in V$, додається ребро (s, v) з вагою $w(s, v) = 0$.

3. Запуск Беллмана-Форда: На цьому модифікованому графі G' запускається алгоритм Беллмана-Форда з початковою вершиною s. Мета цього кроку — обчислити найкоротшу відстань від s до кожної вершини v у G'. Цю відстань позначимо як $h(v)$.

- Виявлення циклів від'ємної ваги: Якщо алгоритм Беллмана-Форда виявляє цикл від'ємної ваги в G', це означає, що оригінальний граф G також містить цикл від'ємної ваги. У такому випадку алгоритм Джонсона завершується, оскільки найкоротші шляхи не визначені.

Крок 2: Переважування ребер графа

1. Використання $h(v)$ для переважування: На основі відстаней $h(v)$, обчислених Беллманом-Фордом на Кроці 1, створюється новий граф G'' з новими вагами ребер.

2. Формула переважування: Для кожного оригінального ребра (u, v) у графі G з оригінальною вагою $w(u, v)$, його нова вага $w'(u, v)$ обчислюється за формулою: $w'(u, v) = w(u, v) + h(u) - h(v)$

3. Властивість невід'ємності: Ключова властивість цього переважування полягає в тому, що всі нові ваги $w'(u, v)$ будуть невід'ємними ($w'(u, v) \geq 0$). Це випливає з принципу трикутника найкоротших шляхів, згідно з яким $h(v) \leq h(u) + w(u, v)$, отже $w(u, v) + h(u) - h(v) \geq 0$. Це перетворює граф з від'ємними вагами на граф лише з невід'ємними вагами, зберігаючи властивості найкоротших шляхів.

4. Збереження найкоротших шляхів: Важливо, що таке переважування не змінює відносний порядок шляхів. Тобто, якщо шлях P_1 був коротшим за шлях P_2 в оригінальному графі, він залишиться коротшим у переваженому графі. Довжина шляху $P = \langle v_0, v_1, \ldots, v_k \rangle$ у переваженому графі буде: $w'(P) = \sum_{i=0}^{k-1} w'(v_i, v_{i+1}) = \sum_{i=0}^{k-1} (w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}))$ $w'(P) = \left(\sum_{i=0}^{k-1} w(v_i, v_{i+1})\right) + h(v_0) - h(v_k) = w(P) + h(v_0) - h(v_k)$ Таким чином, вага шляху в переваженому графі відрізняється від ваги в оригінальному графі на константу $h(v_0) - h(v_k)$, яка залежить лише від початкової та кінцевої вершин шляху, а не від самого шляху. Це гарантує, що найкоротші шляхи зберігаються.

Крок 3: Запуск Дейкстри з кожної вершини на переваженому графі

1. Виконання Дейкстри: Оскільки всі ваги ребер у G'' тепер невід'ємні, ми можемо ефективно використовувати алгоритм Дейкстри.

2. Багаторазовий запуск: Для кожної оригінальної вершини $u \in V$, запускається алгоритм Дейкстри на графі G'' (з новими вагами

$w'(u, v)$) для знаходження найкоротших відстаней від u до всіх інших вершин v. Позначимо ці відстані як $d'(u, v)$.

3. Перетворення результатів: Оскільки $d'(u, v)$ — це найкоротша відстань у переваженому графі, для отримання фактичної найкоротшої відстані $d(u, v)$ в оригінальному графі G, використовується формула: $d(u, v) = d'(u, v) - h(u) + h(v)$

1.2. Псевдокод алгоритму Джонсона

Для реалізації алгоритму Джонсона нам знадобляться реалізації алгоритмів Беллмана-Форда та Дейкстри, описані в попередніх розділах.

Структури даних, що використовуються:

- `Graph G`: Вхідний зважений орієнтований граф.
- `V`: Загальна кількість вершин у графі.
- `dist[V][V]`: Матриця, що зберігатиме кінцеві найкоротші відстані між усіма парами вершин.

```
// Допоміжні операції:

// Операція: КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)
// Призначення: Повертає загальну кількість вершин у Графі.

// Операція: СТВОРИТИ_НОВИЙ_ГРАФ(КІЛЬКІСТЬ_ВЕРШИН)
// Призначення: Створює новий порожній граф з заданою кількістю верши
н.

// Операція: ДОДАТИ_РЕБРО_В_ГРАФ(Граф, ДЖЕРЕЛО, ПРИЗНАЧЕННЯ, ВАГА)
// Призначення: Додає орієнтоване ребро з певною вагою до графа.

// Операція: ОТРИМАТИ_РЕБРА_ГРАФА(Граф)
// Призначення: Повертає список всіх ребер графа (U, V, W).

// Операція: BELLMAN_FORD_ALGORITHM(Graph, start_node)
// Призначення: Алгоритм Беллмана-Форда.
// Повертає: Масив відстаней dist[], або повідомлення про наявність н
егативного циклу.

// Операція: DIJKSTRA_ALGORITHM(Graph, start_node)
// Призначення: Алгоритм Дейкстри.
// Повертає: Масив відстаней dist[].
```

```
// Константа для нескінченності (дуже велике число)
КОНСТАНТА НЕСКІНЧЕННІСТЬ = 999999999

---

АЛГОРИТМ JOHNSON_ALGORITHM(Graph G):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ МАТРИЦЮ final_dist РОЗМІРОМ [КІЛЬКІСТЬ_ВЕРШИН][КІЛЬКІСТЬ_ВЕРШИН] // Для кінцевих результатів APSP

  3. // Крок 1: Додаємо нову вершину 's' та запускаємо Беллмана-Форда
  4. G_prime = СТВОРИТИ_НОВИЙ_ГРАФ(КІЛЬКІСТЬ_ВЕРШИН + 1) // Новий граф з V+1 вершинами (0..V-1 оригінальні, V - нова 's')
  5. ДЛЯ КОЖНОГО РЕБРА (U, V, W) З ОТРИМАТИ_РЕБРА_ГРАФА(G):
      a. ДОДАТИ_РЕБРО_В_ГРАФ(G_prime, U, V, W) // Додаємо всі оригінальні ребра
  6. ДЛЯ КОЖНОЇ ВЕРШИНИ i ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. ДОДАТИ_РЕБРО_В_ГРАФ(G_prime, КІЛЬКІСТЬ_ВЕРШИН, i, 0) // Додаємо ребра (s, i) з вагою 0

  7. h_values = BELLMAN_FORD_ALGORITHM(G_prime, КІЛЬКІСТЬ_ВЕРШИН) // Запускаємо Беллмана-Форда з 's'
  8. ЯКЩО h_values Є_ПОВІДОМЛЕННЯ_ПРО_НЕГАТИВНИЙ_ЦИКЛ:
      a. ПОВЕРНУТИ "Граф містить цикл від'ємної ваги."

  9. // Крок 2: Переважуємо ребра
  10. СТВОРИТИ_НОВИЙ_ГРАФ G_reweighted = СТВОРИТИ_НОВИЙ_ГРАФ(КІЛЬКІСТЬ_ВЕРШИН)
  11. ДЛЯ КОЖНОГО РЕБРА (U, V, W) З ОТРИМАТИ_РЕБРА_ГРАФА(G):
      a. W_prime = W + h_values[U] - h_values[V] // Формула переважування
      b. ДОДАТИ_РЕБРО_В_ГРАФ(G_reweighted, U, V, W_prime)

  12. // Крок 3: Запускаємо Дейкстру з кожної вершини на переваженому графі
  13. ДЛЯ КОЖНОЇ ВЕРШИНИ U ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
      a. d_prime_from_u = DIJKSTRA_ALGORITHM(G_reweighted, U) // Запускаємо Дейкстру
      b. ДЛЯ КОЖНОЇ ВЕРШИНИ V ВІД 0 ДО (КІЛЬКІСТЬ_ВЕРШИН - 1):
          i. ЯКЩО d_prime_from_u[V] == НЕСКІНЧЕННІСТЬ:
              1. final_dist[U][V] = НЕСКІНЧЕННІСТЬ // Якщо шлях не існує
          ii. ІНАКШЕ:
              1. final_dist[U][V] = d_prime_from_u[V] - h_values[U] + h_values[V] // Перетворення назад
```

```
14. ПОВЕРНУТИ final_dist
```
Операції та їх пояснення:

- `КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)`: Повертає загальну кількість вершин V у графі.
- `СТВОРИТИ_НОВИЙ_ГРАФ(КІЛЬКІСТЬ_ВЕРШИН)`: Створює новий, порожній граф з заданою кількістю вершин.
- `ДОДАТИ_РЕБРО_В_ГРАФ(Граф, ДЖЕРЕЛО, ПРИЗНАЧЕННЯ, ВАГА)`: Додає орієнтоване ребро від `ДЖЕРЕЛО` до `ПРИЗНАЧЕННЯ` з вказаною `ВАГОЮ`.
- `ОТРИМАТИ_РЕБРА_ГРАФА(Граф)`: Повертає список всіх ребер графа. Кожне ребро представлене як трійка `(джерело, призначення, вага)`.
- `BELLMAN_FORD_ALGORITHM(Graph, start_node)`: Це виклик раніше описаного алгоритму Беллмана-Форда. Він повинен повернути масив найкоротших відстаней від `start_node` до всіх інших вершин, або ж спеціальне повідомлення, якщо виявлено цикл від'ємної ваги. `h_values` — це результат цього Беллмана-Форда.
- `DIJKSTRA_ALGORITHM(Graph, start_node)`: Це виклик раніше описаного алгоритму Дейкстри. Він повинен повернути масив найкоротших відстаней від `start_node` до всіх інших вершин у графі з невід'ємними вагами. `d_prime_from_u` — це результат цього Дейкстри.
- `НЕСКІНЧЕННІСТЬ`: Символічна константа для представлення недосяжних вершин або нескінченних відстаней.
- `G_prime`: Допоміжний граф, який використовується для запуску Беллмана-Форда. Він включає всі оригінальні вершини та ребра, а також нову вершину s (з індексом `КІЛЬКІСТЬ_ВЕРШИН`) та нульові ребра від s до всіх оригінальних вершин.
- `h_values`: Масив, що зберігає найкоротші відстані від нової вершини s до всіх оригінальних вершин графа. Ці значення використовуються для переважування.

– `ЯКЩО h_values Є_ПОВІДОМЛЕННЯ_ПРО_НЕГАТИВНИЙ_ЦИКЛ`: Перевірка результату Беллмана-Форда. Якщо він виявив цикл від'ємної ваги, то такий цикл існує і в оригінальному графі, тому алгоритм Джонсона завершується.

– `G_reweighted`: Це новий граф, який є копією оригінального графа, але з усіма ребрами, переваженими за формулою $w'(u, v) = w(u, v) + h(u) - h(v)$. Усі ваги в цьому графі будуть невід'ємними.

– `final_dist[U][V]`: Це кінцева матриця, яка після виконання алгоритму міститиме найкоротші відстані від кожної вершини U до кожної вершини V в оригінальному графі.

– Перетворення результатів Дейкстри: Після отримання $d'(u, v)$ (відстаней у переваженому графі), ми використовуємо формулу $d(u, v) = d'(u, v) - h(u) + h(v)$ для перетворення їх назад до оригінальних ваг.

1.3. Часова та просторова складність

Часова складність: Часова складність алгоритму Джонсона є сумою часу, необхідного для кожного кроку:

1. Крок 1 (Беллман-Форд): $O(V \cdot E)$ для графа G' (який має $V + 1$ вершин та $E + V$ ребер). Отже, це $O(V \cdot E)$.

2. Крок 2 (Переважування): $O(V + E)$ для перегляду всіх ребер та обчислення нових ваг.

3. Крок 3 (V разів Дейкстра): Запуск Дейкстри V разів. Кожен запуск Дейкстри з використанням бінарної купи займає $O((V + E)\log V)$ часу. Отже, V запусків займуть $V \cdot O((V + E)\log V)$.

Загальна часова складність: $O(V \cdot E + V \cdot (V + E)\log V)$.

– Для розріджених графів (де $E \approx V$), це спрощується до $O(V^2 \log V)$. Це значно краще, ніж $O(V^3)$ Флойд-Воршелла для розріджених графів.

– Для щільних графів (де $E \approx V^2$), це спрощується до $O(V^3 \log V)$ або навіть $O(V^4)$ залежно від точної імплементації черги з пріоритетом. У цьому випадку Флойд-Воршелл може бути конкурентоспроможним або навіть швидшим.

Просторова складність: Просторова складність алгоритму Джонсона становить $O(V^2)$ для зберігання кінцевої матриці `final_dist`, а також $O(V + E)$ для зберігання графа та проміжних масивів Дейкстри/Беллмана-Форда. Таким чином, домінує $O(V^2 + E)$, що фактично є $O(V^2)$ для переважної більшості графів, де $E \leq V^2$.

3. Область застосування

Алгоритм Джонсона є оптимальним вибором для знаходження найкоротших шляхів між усіма парами вершин у зважених орієнтованих графах, коли:

1. Граф може містити ребра з від'ємними вагами, але без циклів від'ємної ваги.
2. Граф є розрідженим (тобто, кількість ребер E значно менша за V^2). Це його основна перевага над Флойд-Воршеллом.
3. Потрібні всі пари найкоротших шляхів, а не лише від одного джерела.
4. Важливо виявити наявність циклів від'ємної ваги (через крок Беллмана-Форда).

Типові області застосування:

– Маршрутизація в мережах: Особливо в тих, де вартість зв'язку може мати від'ємні значення (наприклад, бонуси за використання певних маршрутів), і потрібно знати всі можливі шляхи.

– Комп'ютерна біологія: У задачах вирівнювання послідовностей ДНК або білків, де "штрафи" за невідповідності можуть моделюватися як від'ємні ваги.

– Дослідження операцій: У задачах планування та оптимізації, де ваги можуть представляти витрати/вигоди, і потрібно знайти всі оптимальні зв'язки між станами.

– Графові бази даних: Для складних запитів про зв'язки між усіма елементами.

3. Приклади застосування

– Оптимізоване планування транспортних мереж (з бонусами): Уявіть транспортну мережу, де деякі маршрути не тільки коштують грошей

(позитивна вага), а й можуть надавати “бонуси” (наприклад, через державні субсидії за використання певних видів транспорту або маршрутів, що веде до “від’ємної вартості” за проїзд). Якщо ця мережа розріджена (не всі точки з’єднані безпосередньо з усіма іншими), і вам потрібно знати найдешевший маршрут між *будь-якою парою* точок, алгоритм Джонсона буде ідеальним. Він зможе знайти найоптимальніші шляхи, враховуючи ці бонуси, і повідомить, якщо існує можливість нескінченного “заробітку” через цикл від’ємної ваги.

- Аналіз мікросхем для виявлення затримок: У дизайні мікросхем, граф може моделювати логічні елементи (вершини) та зв’язки між ними (ребра). Ваги ребер можуть представляти затримки сигналу. Деякі затримки можуть бути від’ємними (наприклад, через “форсування” сигналу). Якщо потрібно знати затримку сигналу між *будь-якою парою* точок на чипі, і граф є розрідженим, алгоритм Джонсона може бути використаний. Він також виявить, чи існують “аномалії” (цикли від’ємної затримки), які можуть вказувати на нестабільність або помилки в дизайні.
- Побудова словника найкоротших шляхів у великих, але розріджених базах даних: У деяких графових базах даних, які моделюють відносини (наприклад, між об’єктами або користувачами), може виникнути потреба обчислювати найкоротші шляхи між усіма парами об’єктів для швидких запитів. Якщо кількість об’єктів велика, але зв’язків між ними відносно мало (розріджений граф), і деякі “відносини” можуть мати від’ємну “вагу” (наприклад, зменшувати відстань або вартість взаємодії), алгоритм Джонсона може бути використаний для попереднього обчислення та зберігання цих шляхів.

Переваги та Недоліки

Переваги алгоритму Джонсона:

- Обробка від’ємних ваг ребер: Головна перевага полягає в тому, що він коректно працює з графами, які містять від’ємні ваги ребер, що відрізняє його від алгоритму Дейкстри.

– Виявлення циклів від'ємної ваги: Завдяки включенню Беллмана-Форда як першого кроку, алгоритм Джонсона здатний ефективно виявляти наявність циклів від'ємної ваги.

– Ефективність для розріджених графів: Для розріджених графів ($E \ll V^2$) його часова складність ($O(VE + V^2 \log V)$ або $O(VE + V(V + E)\log V)$) є кращою, ніж у Флойд-Воршелла ($O(V^3)$).

– Використання перевірених алгоритмів: Він базується на добре вивчених і реалізованих алгоритмах Беллмана-Форда та Дейкстри.

Недоліки алгоритму Джонсона:

– Складність реалізації: Він складніший у реалізації порівняно з алгоритмом Флойд-Воршелла, оскільки вимагає реалізації двох різних алгоритмів (Беллмана-Форда та Дейкстри) і кроку переважування.

– Найгірший випадок: Для щільних графів ($E \approx V^2$) його часова складність може бути гіршою за Флойд-Воршелла, наближаючись до $O(V^3 \log V)$.

– Додаткова пам'ять: Потребує додаткової пам'яті для зберігання допоміжного графа G' та масиву $h(v)$.

– Не знаходить шляхи за наявності циклів від'ємної ваги: Як і Беллман-Форд, він лише повідомляє про наявність такого циклу, але не може знайти "найкоротші" шляхи, оскільки вони прямують до $-\infty$.

Контрольні Запитання

1. Призначення Джонсона: Яку проблему вирішує алгоритм Джонсона? Які типи ваг ребер він може обробляти?

2. Основні кроки: Опишіть три основні кроки алгоритму Джонсона.

3. Роль Беллмана-Форда: Для чого використовується алгоритм Беллмана-Форда на першому кроці алгоритму Джонсона? Яку інформацію він надає?

4. Переважування ребер: Поясніть формулу переважування ребер $w'(u, v) = w(u, v) + h(u) - h(v)$. Чому після переважування всі ваги стають невід'ємними?

5. Збереження шляхів: Чому переважування ребер не змінює найкоротші шляхи (лише їхню вагу)?
6. Роль Дейкстри: Для чого використовується алгоритм Дейкстри на третьому кроці алгоритму Джонсона?
7. Виявлення циклів від'ємної ваги: Як алгоритм Джонсона виявляє наявність циклу від'ємної ваги?
8. Часова складність: Яка часова складність алгоритму Джонсона? Як вона відрізняється для розріджених та щільних графів?
9. Порівняння з Флойд-Воршеллом: Коли алгоритм Джонсона є кращим вибором, ніж алгоритм Флойд-Воршелла, і чому?
10. Застосування: Наведіть один приклад реальної задачі, де алгоритм Джонсона є корисним.

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота №X: Реалізація Алгоритму Джонсона для Знаходження Найкоротших Шляхів Між Усіма Парами Вершин з Від'ємними Вагами

2. Мета

Метою даної лабораторної роботи є набуття практичних навичок у реалізації алгоритму Джонсона для знаходження найкоротших шляхів між усіма парами вершин у зважених орієнтованих графах, що містять від'ємні ваги ребер, а також у виявленні циклів від'ємної ваги.

3. Завдання

a) Реалізація представлення графа: Створити клас або структуру для представлення орієнтованого зваженого графа (рекомендується списки суміжності для Дейкстри, або список ребер для Беллмана-Форда).
b) Реалізація Беллмана-Форда: Використати або реалізувати функціональну версію алгоритму Беллмана-Форда, яка повертає масив відстаней або повідомлення про цикл від'ємної ваги.

c) Реалізація Дейкстри: Використати або реалізувати функціональну версію алгоритму Дейкстри (з чергою з пріоритетом), яка працює зі зваженими графами з невід'ємними вагами.

d) Реалізація алгоритму Джонсона: Розробити функцію `johnson(Graph G)`, яка інтегрує Беллмана-Форда та Дейкстру для розв'язання APSP.

e) Відновлення шляху: Реалізувати функцію `reconstructPath(int[][] predecessor, int start_node, int target_node)`, яка відновлює та виводить найкоротший шлях.

f) Тестування: Протестувати реалізований алгоритм на різних зважених орієнтованих графах, включаючи випадки, де:

- Граф містить позитивні ваги.
- Граф містить від'ємні ваги ребер, але без циклів від'ємної ваги.
- Граф містить цикл від'ємної ваги.
- Граф є незв'язним.

4. Опис

Вам необхідно розробити програмний додаток на обраній мові програмування (наприклад, Python, C++, Java, JavaScript). Додаток повинен дозволяти:

– Створювати орієнтовані зважені графи з заданою кількістю вершин та ребер.

– Додавати ребра з вагами (включаючи від'ємні) до графа.

– Викликати функцію `johnson` для обчислення всіх найкоротших шляхів.

– Виводити отриману матрицю найкоротших відстаней.

– Виводити відновлений найкоротший шлях між заданими парами вершин.

– Чітко повідомляти, якщо виявлено цикл від'ємної ваги.

Для представлення графа в цій лабораторній роботі рекомендується використовувати списки суміжності, оскільки це більш природно для алгоритму Дейкстри.

5. Порядок Виконання

a) Підготовка:

- Створіть новий проект.
- Реалізуйте клас `WeightedDirectedGraph` з використанням списків суміжності, який дозволяє додавати ребра з вагами.
- Реалізуйте або інтегруйте функції `bellmanFord(Graph G, int start_node)` та `dijkstra(Graph G, int start_node)`. Переконайтеся, що `bellmanFord` може повернути індикатор циклу від'ємної ваги.

a) Реалізація алгоритму Джонсона:

- Створіть функцію `johnson(Graph G)`.
- Реалізуйте Крок 1: створення G', додавання s та нульових ребер, запуск Беллмана-Форда. Обробіть випадок негативного циклу.
- Реалізуйте Крок 2: створення G'' з переваженими ребрами.
- Реалізуйте Крок 3: запуск Дейкстри для кожної вершини на G'' та перетворення відстаней назад.

b) Реалізація відновлення шляху:

- Створіть функцію `reconstructPath(int[][] predecessor_matrix, int start_node, int target_node)`, яка може використовувати матрицю попередників, отриману з кожного запуску Дейкстри (або модифікуйте Дейкстру, щоб вона повертала попередників).

c) Тестування:

- Створіть кілька екземплярів `Graph` для різних тестових сценаріїв (як зазначено в Завданнях, пункт 6).
- Для кожного графа:

        - Викличте `johnson`.
        - Виведіть матрицю відстаней.
        - Якщо немає циклу від'ємної ваги, викличте `reconstructPath` для кількох пар `(start_node, target_node)` і виведіть відновлені шляхи.
        - Чітко виведіть повідомлення про виявлення циклу від'ємної ваги, якщо він є.

5. Результат

Після виконання всіх завдань програми повинні виводити:

– Для кожного тестового графа:

- Або матрицю найкоротших відстаней.
- Або повідомлення про наявність циклу від'ємної ваги.
- Якщо шляхи визначені, для кожної заданої пари `(start_node, target_node)`: відновлений найкоротший шлях.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:
    - Реалізовано структуру даних для представлення орієнтованого зваженого графа (зі списками суміжності).
    - Надано функціональну реалізацію алгоритмів Беллмана-Форда та Дейкстри (як допоміжних для Джонсона).
    - Надано функціональну реалізацію алгоритму Джонсона, включаючи логіку виявлення циклів від'ємної ваги.
    - Надано функцію для відновлення найкоротшого шляху між будь-якою парою вершин.
    - Код має бути добре структурованим, читабельним та містити необхідні коментарі.
2. Продемонструвати роботу додатку:

- Продемонструвати роботу алгоритму Джонсона на 3-4 прикладах графів, що ілюструють різні випадки:
    - Граф з позитивними вагами.
    - Граф з від'ємними вагами без циклів.
    - Граф з циклом від'ємної ваги.
    - Незв'язні графи.
- Показати коректність виводу матриці відстаней та відновлених шляхів, або чітке повідомлення про наявність циклу від'ємної ваги.

3. Бути готовим відповісти на питання:
    - Принцип роботи: Детально пояснити покрокову роботу алгоритму Джонсона, включаючи роль Беллмана-Форда, процес переважування та багаторазовий запуск Дейкстри.
    - Переважування: Пояснити, як і чому переважування ребер робить їх невід'ємними, і чому це не спотворює найкоротші шляхи.
    - Виявлення циклів: Пояснити, як алгоритм Джонсона виявляє цикли від'ємної ваги.
    - Складність: Обґрунтувати часову та просторову складність алгоритму Джонсона.
    - Порівняння: Порівняти Джонсона з Флойд-Воршеллом, Дейкстрою (V разів) та Беллманом-Фордом (V разів), вказавши, коли кожен з них є найбільш доцільним.
    - Застосування: Навести 2-3 приклади реальних задач, де алгоритм Джонсона є корисним.

Тести

1. Які два алгоритми комбінує алгоритм Джонсона?

    А) BFS та DFS.

    Б) Дейкстри та Флойд-Воршелла.

В) Беллмана-Форда та Дейкстри.

Г) Прима та Крускала.

2. Яку проблему вирішує алгоритм Джонсона?

   А) Найкоротший шлях від одного джерела.

   Б) Мінімальне остовне дерево.

   В) Найкоротші шляхи між усіма парами вершин.

   Г) Топологічне сортування.

3. Яке обмеження на ваги ребер має оригінальний граф для коректної роботи алгоритму Джонсона?

   А) Тільки позитивні ваги.

   Б) Тільки від'ємні ваги.

   В) Від'ємні ваги дозволені, але без циклів від'ємної ваги.

   Г) Немає обмежень на ваги.

4. Яка часова складність алгоритму Джонсона для розріджених графів?

   А) $O(V^3)$

   Б) $O(VE)$

   В) $O(V^2 \log V)$

   Г) $O(V \cdot E \log V)$

5. Для чого використовується Беллман-Форд на першому кроці алгоритму Джонсона?

   А) Для обчислення кінцевих найкоротших відстаней.

   Б) Для виявлення циклів від'ємної ваги та обчислення значень переважування.

   В) Для побудови мінімального остовного дерева.

   Г) Для топологічного сортування.

Тести підвищеної складності

1. Чому у переваженому графі G'' (після Кроку 2) всі ваги ребер стають невід'ємними?

   А) Тому що всі від'ємні ваги просто замінюються на 0.

   Б) Тому що це випливає з властивостей найкоротших шляхів та трикутника нерівностей після застосування Беллмана-Форда.

   В) Тому що це обов'язкова умова для запуску Беллмана-Форда.

   Г) Тому що алгоритм Дейкстри не може працювати з від'ємними вагами.

2. Яке твердження вірно щодо часової складності алгоритму Джонсона порівняно з алгоритмом Флойд-Воршелла?

   А) Джонсон завжди швидший.

   Б) Флойд-Воршелл завжди швидший.

   В) Джонсон швидший для розріджених графів, Флойд-Воршелл кращий для щільних.

   Г) Їхня часова складність однакова.

3. Яка роль нової вершини s та нульових ребер з неї на першому кроці алгоритму Джонсона?

   А) Зробити граф сильно зв'язним.

   Б) Гарантувати, що Беллман-Форд зможе обчислити найкоротші шляхи до всіх оригінальних вершин.

   В) Видалити всі цикли від'ємної ваги.

   Г) Зменшити загальну кількість ребер у графі.

4. Як обчислюється фактична найкоротша відстань $d(u, v)$ в оригінальному графі за результатами $d'(u, v)$ у переваженому графі?

   А) $d(u, v) = d'(u, v) + h(u) + h(v)$

   Б) $d(u, v) = d'(u, v) - h(u) - h(v)$

   В) $d(u, v) = d'(u, v) - h(u) + h(v)$

Г) $d(u, v) = d'(u, v) + h(u) - h(v)$

5. Який алгоритм запускається з кожної вершини на переваженому графі на третьому кроці алгоритму Джонсона?

   А) BFS.

   Б) DFS.

   В) Алгоритм Беллмана-Форда.

   Г) Алгоритм Дейкстри.

6. Якщо Беллман-Форд на першому кроці алгоритму Джонсона виявляє цикл від'ємної ваги, що відбувається далі?

   А) Алгоритм Джонсона продовжує роботу, ігноруючи цикл.

   Б) Алгоритм Джонсона повертає повідомлення про наявність циклу від'ємної ваги і завершує роботу.

   В) Алгоритм Джонсона намагається видалити цикл від'ємної ваги.

   Г) Алгоритм Джонсона автоматично переходить до Флойд-Воршелла.

7. Чому алгоритм Джонсона може бути кращим вибором, ніж запуск Беллмана-Форда V разів, для розріджених графів з від'ємними вагами?

   А) Беллман-Форд не працює з від'ємними вагами.

   Б) Запуск Беллмана-Форда V разів буде мати складність $O(V^2E)$, що зазвичай гірше, ніж $O(VE + V^2 \log V)$.

   В) Джонсон не вимагає виявлення циклів від'ємної ваги.

   Г) Беллман-Форд не може обчислювати значення переважування.

8. Яка перевага алгоритму Джонсона над алгоритмом Дейкстри?

   А) Джонсон має меншу просторову складність.

   Б) Джонсон може обробляти від'ємні ваги ребер.

   В) Джонсон завжди швидший.

   Г) Джонсон може працювати з неорієнтованими графами.

9. Яке застосування НЕ є типовим для алгоритму Джонсона?

   А) Маршрутизація в мережах з бонусами.

   Б) Вирівнювання послідовностей ДНК з штрафами.

   В) Знаходження найкоротшого шляху від одного джерела у графі з від'ємними вагами.

   Г) Виявлення аномалій (циклів від'ємної затримки) у дизайні мікросхем.

10. Яка часова складність алгоритму Джонсона для щільних графів?

    А) $O(V^2 \log V)$

    Б) $O(V^3)$

    В) $O(V^3 \log V)$

    Г) $O(VE)$

Завдання для самостійної роботи

1. Повна реалізація: Реалізуй весь алгоритм Джонсона, інтегруючи свої реалізації Беллмана-Форда та Дейкстри. Перевір його роботу на кількох тестових графах, включаючи випадки з позитивними, від'ємними вагами та негативними циклами.

2. Візуалізація переважування: Створи простий інструмент для візуалізації кроку переважування: показуй оригінальний граф, виведи $\mathsf{h}(\mathsf{v})$ для кожної вершини, потім покажи переважений граф з новими вагами.

3. Порівняння ефективності: Порівняй час виконання алгоритму Джонсона з алгоритмами Флойд-Воршелла та Беллмана-Форда (запущеним V разів) на різних типах графів (розріджені, щільні) з від'ємними вагами. Побудуй графіки залежності часу від кількості вершин.

4. Відновлення шляхів: Реалізуй функцію, яка може відновити повний найкоротший шлях між будь-якою парою вершин, використовуючи результати алгоритму Джонсона.

5. Транзитивне замикання: Адаптуй алгоритм Джонсона для знаходження транзитивного замикання графа.
6. Змінені ваги: Досліди, як алгоритм Джонсона може бути використаний для ефективного переобчислення найкоротших шляхів після зміни ваги одного ребра.
7. Імплементація з різних мовних бібліотек: Якщо дозволено, використай вбудовані бібліотеки для Беллмана-Форда та Дейкстри, щоб зосередитись на логіці інтеграції Джонсона.
8. Застосування в арбітражі: Змоделюй невелику мережу обміну валют (з від'ємними логарифмічними вагами для прибутку/збитку) та застосуй Джонсона для виявлення можливостей арбітражу між усіма парами валют.
9. Оптимізація пам'яті: Якщо граф дуже великий, але розріджений, подумай, як можна оптимізувати пам'ять, щоб не зберігати повністю `final_dist` матрицю, якщо шляхи потрібні лише за запитом.
10. Алгоритм Джонсона для неорієнтованих графів: Поясни, як алгоритм Джонсона може бути застосований до неорієнтованих графів, та які зміни потрібні.

Відповіді: 1 - В), 2 - В), 3 - В), 4 - В), 5 - Б).

Відповіді:

1 Б). Тому що це випливає з властивостей найкоротших шляхів та трикутника нерівностей після застосування Беллмана-Форда.

2 В). Джонсон швидший для розріджених графів, Флойд-Воршелл кращий для щільних.

3 Б). Гарантувати, що Беллман-Форд зможе обчислити найкоротші шляхи до всіх оригінальних вершин.

4 В). $d(u, v) = d'(u, v) - h(u) + h(v)$.

5 Г). Алгоритм Дейкстри.

6 Б). Алгоритм Джонсона повертає повідомлення про наявність циклу від'ємної ваги і завершує роботу.

7 Б). Запуск Беллмана-Форда V разів буде мати складність $O(V^2E)$, що зазвичай гірше, ніж $O(VE + V^2 \log V)$.

8 Б). Джонсон може обробляти від'ємні ваги ребер.

9 В). Знаходження найкоротшого шляху від одного джерела у графі з від'ємними вагами.

10 В). $O(V^3 \log V)$.

Евристичний пошук: A*.

Евристичний пошук: розумне розв'язання складних задач

У попередніх розділах ми з вами ознайомились із фундаментальними алгоритмами пошуку шляхів у графах, такі як BFS та Дейкстра. Ці алгоритми гарантовано знаходять найкоротші шляхи, але вони можуть бути досить "сліпими" у своєму розширенні: вони досліджують усі можливі напрямки, доки не знайдуть оптимальне рішення.

У великих, складних графах або просторах станів такий "сліпий" пошук може бути надзвичайно неефективним, споживаючи величезні обсяги часу та пам'яті. Уявіть собі спробу знайти найкоротший маршрут на Google Maps, перевіряючи кожне можливе перехрестя на кожному рівні — це було б неможливо.

Саме тут на допомогу приходить концепція евристичного пошуку. Евристичний пошук — це клас алгоритмів, які використовують додаткову, "розумну" інформацію (називається евристикою) для керівництва своїм процесом пошуку. Замість того, щоб розширювати вузли без розбору, евристичний пошук намагається оцінити, наскільки "перспективним" є кожен вузол для досягнення цілі. Це дозволяє алгоритму "цілеспрямовано" рухатися до мети, уникаючи дослідження завідомо менш перспективних гілок.

Евристика — це, по суті, "правило великого пальця" або "освічене припущення" щодо того, наскільки близько ми знаходимося до розв'язку. Вона не обов'язково є абсолютно точною, але повинна надавати достатньо хорошої оцінки, щоб значно прискорити пошук.

Наприклад, у навігації, евристикою може бути пряма відстань “по повітрю” до цільового пункту. Це не справжня відстань по дорогах, але вона дає уявлення про те, наскільки далеко знаходиться ціль.

Перевага евристичного пошуку полягає в тому, що він дозволяє розв’язувати задачі, які були б обчислювально нерозв’язними для “сліпих” алгоритмів. Він знаходить баланс між гарантією оптимальності (яку надає Дейкстра) та швидкістю (яку надає жадібний пошук). Хоча не всі евристичні алгоритми гарантують знаходження оптимального рішення, ті, що мають певні властивості евристики, можуть це робити.

У цьому розділі ми детально розглянемо один із найвідоміших та найефективніших алгоритмів евристичного пошуку — алгоритм A*. Він майстерно поєднує переваги Дейкстри з потужністю евристичної інформації, дозволяючи нам знаходити оптимальні шляхи у складних та великих просторах станів з дивовижною швидкістю.

Алгоритм A* — це алгоритм пошуку найкоротшого шляху, який знаходить шлях з найменшою вартістю від початкової вершини до цільової у зваженому графі. Він був розроблений у 1968 році.

A* є інформованим алгоритмом пошуку, що означає, що він використовує додаткову “евристичну” інформацію для керівництва своїм пошуком. Ця евристика допомагає алгоритму приймати “розумніші” рішення щодо того, яку вершину відвідати наступною, значно прискорюючи пошук порівняно з неінформованими алгоритмами (наприклад, BFS або Дейкстра без оптимізації).

A* поєднує в собі найкращі риси жадібного пошуку за найкращою першою вершиною та алгоритму Дейкстри:

- Як Дейкстра, A* є оптимальним (знаходить найкоротший шлях) за певних умов.
- Як жадібний пошук, A* є ефективним (швидко знаходить шлях) завдяки евристиці.

1.1. Функція оцінки $f(n) = g(n) + h(n)$

Алгоритм A* працює, оцінюючи “перспективність” кожної вершини n за допомогою функції оцінки $f(n)$, яка складається з двох компонентів:

1. $g(n)$ (вартість шляху від старту): Це фактична вартість (вага) найкоротшого шляху, знайденого від початкової вершини до

поточної вершини n. Якщо граф незважений, $g(n)$ буде просто кількістю ребер. Якщо граф зважений, $g(n)$ — це сума ваг ребер на шляху. Ця частина функції робить A* схожим на Дейкстру, забезпечуючи пошук оптимального шляху.

2. $h(n)$ (евристична вартість до мети): Це оцінка (прогнозована вартість) найкоротшого шляху від поточної вершини n до цільової вершини. Це "евристика" — наближене значення, яке не обов'язково є точним, але має бути "розумним" припущенням. Наприклад, для навігації на карті евристикою може бути пряма відстань до мети (евклідова або манхеттенська). Ця частина функції робить A* схожим на жадібний пошук, спрямовуючи його до мети.

Алгоритм A* завжди розширює ту вершину, яка має найменше значення $f(n)$ у черзі з пріоритетом.

1.2. Властивості евристичної функції: допустимість та узгодженість

Якість та властивості евристичної функції $h(n)$ є критично важливими для продуктивності та коректності алгоритму A*.

1. Допустимість

   - Евристика $h(n)$ називається допустимою, якщо для кожної вершини n вона ніколи не переоцінює фактичну вартість (найкоротший шлях) від n до цільової вершини. Тобто, $h(n) \leq$ true_cost(n,goal), де true_cost(n,goal) — це фактична вага найкоротшого шляху від n до цільової вершини.

   - Значення: Якщо використовується допустима евристика, алгоритм A* гарантовано знайде оптимальний (найкоротший) шлях. Якщо евристика переоцінює вартість, алгоритм може пропустити оптимальний шлях, вважаючи його "дорожчим", ніж є насправді.

2. Узгодженість

   - Евристика $h(n)$ називається узгодженою (або монотонною), якщо для кожної вершини n та її будь-якого сусіда n': $h(n) \leq \text{cost}(n, n') + h(n')$ Де $\text{cost}(n, n')$ — це фактична вартість переходу від n до n'. Ця умова схожа на нерівність трикутника.

- Значення: Узгодженість є сильнішою умовою, ніж допустимість. Якщо евристика узгоджена, то вона також є допустимою.
- Переваги узгодженості:
  - Забезпечує, що функція $f(n)$ не зменшується вздовж будь-якого шляху від початкової вершини.
  - Якщо евристика узгоджена, алгоритму A* не потрібно повторно обробляти вершини, які вже були "закриті" (переміщені в closed set). Це схоже на поведінку алгоритму Дейкстри, де найкоротший шлях до вершини, щойно вилученої з пріоритетної черги, є остаточним. Проте, "закритий список" все ще зазвичай використовується для підвищення ефективності, запобігаючи повторній обробці вже завершених вершин.

Алгоритм A* працює за принципом, дуже схожим на алгоритм Дейкстри, але з ключовою відмінністю в пріоритезації вершин у черзі. Він використовує дві основні множини вершин:

1. Відкритий список (Open Set / Frontier): Це черга з пріоритетом, яка зберігає вершини, які були відвідані, але їхні сусіди ще не були повністю досліджені. Вершини сортуються за значенням $f(n)$, що дозволяє завжди вибирати найбільш "перспективну" вершину.
2. Закритий список (Closed Set / Visited): Це множина, яка зберігає вершини, які були повністю оброблені, тобто для яких знайдено остаточний найкоротший шлях, і всі їхні сусіди були досліджені.

Покроковий опис алгоритму A*:

1. Ініціалізація:
   - Створити масив `g_score` (або `distance`) розміром V. `g_score[n]` буде зберігати поточну відому вартість найкоротшого шляху від початкової вершини до n. Ініціалізувати `g_score[start_node] = 0` і `g_score[i] = НЕСКІНЧЕННІСТЬ` для всіх інших вершин.
   - Створити масив `f_score` (або `total_cost`) розміром V. `f_score[n]` буде зберігати `g_score[n] + h(n)`.

Ініціалізувати `f_score[start_node] = h(start_node)` і `f_score[i] = НЕСКІНЧЕННІСТЬ` для всіх інших.

- Створити масив `predecessor` (або `came_from`) розміром V, ініціалізувати всі як `NULL` або `-1`. Це дозволить відновити шлях.
- Створити чергу з пріоритетом `open_set`. Додати до `open_set` початкову вершину: `(f_score[start_node], start_node)`.
- Створити множину `closed_set` (або булевий масив `visited_finalized`) для зберігання вершин, які вже були повністю оброблені.

2. Основний цикл:
    - Поки `open_set` не порожній:
        - Вилучити з `open_set` пару `(f_current, current_node)` з найменшим значенням `f_current`. Це вершина, яка, за оцінкою, є найперспективнішою.
        - Якщо `current_node` є цільовою вершиною:
            - Шлях знайдено! Відновити шлях за допомогою масиву `predecessor` і завершити алгоритм.
        - Додати `current_node` до `closed_set`. (Якщо евристика узгоджена, ця перевірка не є строго необхідною для коректності, але покращує продуктивність, запобігаючи повторній обробці).
        - Для кожного сусіда `neighbor` вершини `current_node`:
            - Нехай `cost_edge = вага_ребра(current_node, neighbor)`.
            - Якщо `neighbor` перебуває у `closed_set`, пропустити його (ми вже знайшли найкоротший шлях до нього, або він вже був оброблений).
            - Обчислити потенційну вартість `g_tentative` до `neighbor` через `current_node`: `g_tentative = g_score[current_node] + cost_edge`.

- Якщо `g_tentative < g_score[neighbor]` (знайдено коротший шлях до `neighbor`):
    - Оновити `predecessor[neighbor] = current_node`.
    - Оновити `g_score[neighbor] = g_tentative`.
    - Оновити `f_score[neighbor] = g_score[neighbor] + h(neighbor)`.
    - Якщо `neighbor` не перебуває в `open_set`:
        - Додати `(f_score[neighbor], neighbor)` до `open_set`.
    - (Якщо `neighbor` вже в `open_set`, його пріоритет у черзі буде автоматично оновлений, або потрібно буде додати новий запис, а старий ігнорувати за тією ж логікою, що й у Дейкстрі).

3. Завершення: Якщо `open_set` стає порожнім, і цільова вершина не була досягнута, це означає, що шляху до цільової вершини не існує.

1.4. Псевдокод алгоритму A*

Структури даних, що використовуються:

- `Graph G`: Вхідний зважений граф з невід'ємними вагами ребер.
- `start_node`: Початкова вершина.
- `goal_node`: Цільова вершина.
- `h(n)`: Евристична функція, яка оцінює вартість від вершини `n` до `goal_node`.
- `g_score`: Масив розміром `V`, `g_score[n]` зберігає вартість шляху від `start_node` до `n`.
- `f_score`: Масив розміром `V`, `f_score[n]` зберігає `g_score[n] + h(n)`.
- `predecessor`: Масив розміром `V`, `predecessor[n]` зберігає попередню вершину на найкоротшому шляху до `n`.

- `PriorityQueue open_set`: Черга з пріоритетом, що зберігає пари `(оцінка_f, вершина)`.
- `Set closed_set`: Множина (або булевий масив) для зберігання повністю оброблених вершин.

```
// Допоміжні операції:

// Операція: КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)
// Призначення: Повертає загальну кількість вершин у Графі.

// Операція: ІНІЦІАЛІЗУВАТИ_МАСИВ(Масив, Значення)
// Призначення: Встановлює всі елементи Масиву на задане Значення.

// Операція: ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(PQ, Елемент)
// Призначення: Додає Елемент (пріоритет, значення) до черги.

// Операція: ВИЛУЧИТИ_МІНІМАЛЬНИЙ_З_ЧЕРГИ_З_ПРІОРИТЕТОМ(PQ)
// Призначення: Вилучає та повертає елемент з найменшим пріоритетом з
черги.

// Операція: ЧЕРГА_З_ПРІОРИТЕТОМ_НЕ_ПОРОЖНЯ(PQ)
// Призначення: Перевіряє, чи черга з пріоритетом містить елементи.
// Повертає: true, якщо PQ не порожня, false інакше.

// Операція: ДОДАТИ_В_МНОЖИНУ(Множина, Елемент)
// Призначення: Додає Елемент до Множини.

// Операція: МНОЖИНА_МІСТИТЬ(Множина, Елемент)
// Призначення: Перевіряє, чи Множина містить Елемент.
// Повертає: true, якщо Множина містить Елемент, false інакше.

// Операція: ОТРИМАТИ_СУСІДІВ_ТА_ВАГИ(Graph G, Вершина U)
// Призначення: Повертає список пар (сусідня_вершина, вага_ребра) для
всіх вихідних ребер з Вершини U.

// Операція: h(Вершина n, Цільова_вершина goal_node)
// Призначення: Обчислює евристичну оцінку вартості від n до goal_nod
e.
// Деталізація: Ця функція має бути реалізована згідно з конкретною п
роблемою.
//              Наприклад, для координатних графів - Евклідова або Ман
хеттенська відстань.

// Константа для нескінченності (дуже велике число)
```

```
КОНСТАНТА НЕСКІНЧЕННІСТЬ = 999999999


АЛГОРИТМ A_STAR_SEARCH(Graph G, start_node, goal_node):
  1. КІЛЬКІСТЬ_ВЕРШИН = КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(G)
  2. СТВОРИТИ МАСИВ g_score РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  3. СТВОРИТИ МАСИВ f_score РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  4. СТВОРИТИ МАСИВ predecessor РОЗМІРОМ З КІЛЬКІСТЬ_ВЕРШИН
  5. СТВОРИТИ PriorityQueue open_set
  6. СТВОРИТИ Set closed_set

  7. ІНІЦІАЛІЗУВАТИ_МАСИВ(g_score, НЕСКІНЧЕННІСТЬ) // Відстані від ст
арту до всіх вершин - нескінченність
  8. ІНІЦІАЛІЗУВАТИ_МАСИВ(f_score, НЕСКІНЧЕННІСТЬ) // Загальні оцінки
 - нескінченність
  9. ІНІЦІАЛІЗУВАТИ_МАСИВ(predecessor, NULL) // Попередників немає

  10. g_score[start_node] = 0 // Відстань від старту до старту - 0
  11. f_score[start_node] = h(start_node, goal_node) // f(start) = g(
start) + h(start) = 0 + h(start)
  12. ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(open_set, (f_score[start_node], st
art_node)) // Додаємо стартову вершину в open_set

  13. ПОКИ ЧЕРГА_З_ПРІОРИТЕТОМ_НЕ_ПОРОЖНЯ(open_set):
      a. (f_current, current_node) = ВИЛУЧИТИ_МІНІМАЛЬНИЙ_З_ЧЕРГИ_З_П
РІОРИТЕТОМ(open_set) // Вилучаємо найбільш перспективну вершину

      b. ЯКЩО current_node == goal_node: // Якщо поточна вершина - ці
льова, шлях знайдено
          i. ПОВЕРНУТИ predecessor // Повертаємо масив попередників д
ля відновлення шляху

      c. ЯКЩО МНОЖИНА_МІСТИТЬ(closed_set, current_node): // Якщо верш
ина вже була повністю оброблена
          i. ПРОДОВЖИТИ // Пропускаємо

      d. ДОДАТИ_В_МНОЖИНУ(closed_set, current_node) // Додаємо поточн
у вершину до закритого списку

      e. ДЛЯ КОЖНОЇ ПАРИ (neighbor, cost_edge) З ОТРИМАТИ_СУСІДІВ_ТА_
ВАГИ(G, current_node): // Для кожного сусіда
          i. // Обчислюємо потенційну вартість шляху до сусіда через
 current_node
          ii. g_tentative = g_score[current_node] + cost_edge
```

```
          iii. ЯКЩО g_tentative < g_score[neighbor]: // Якщо знайдено коротший шлях до сусіда
               1. predecessor[neighbor] = current_node // Оновлюємо попередника
               2. g_score[neighbor] = g_tentative // Оновлюємо g_score
               3. f_score[neighbor] = g_score[neighbor] + h(neighbor, goal_node) // Оновлюємо f_score
               4. ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(open_set, (f_score[neighbor], neighbor)) // Додаємо сусіда в open_set (або оновлюємо пріоритет)

  14. ПОВЕРНУТИ NULL // Шляху не існує
```

Операції та їх пояснення:

- `КІЛЬКІСТЬ_ВЕРШИН_В_ГРАФІ(Граф)`: Ця операція повертає загальну кількість вершин у графі. Використовується для ініціалізації масивів.
- `ІНІЦІАЛІЗУВАТИ_МАСИВ(Масив, Значення)`: Допоміжна операція, що заповнює масив заданим значенням.
- `НЕСКІНЧЕННІСТЬ`: Символічна константа для початкових відстаней до недосяжних вершин.
- `ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(...)` / `ВИЛУЧИТИ_МІНІМАЛЬНИЙ_З_ЧЕРГИ_З_ПРІОРИТЕТОМ(...)` / `ЧЕРГА_З_ПРІОРИТЕТОМ_НЕ_ПОРОЖНЯ(...)`: Це стандартні операції з чергою з пріоритетом. Вона використовується для ефективного вибору вершини з найменшим $f(n)$.
- `ДОДАТИ_В_МНОЖИНУ(...)` / `МНОЖИНА_МІСТИТЬ(...)`: Операції з множиною (`closed_set`), що дозволяють швидко перевіряти та додавати вершини до закритого списку.
- `ОТРИМАТИ_СУСІДІВ_ТА_ВАГИ(G, U)`: Повертає список пар `(сусідня_вершина, вага_ребра)` для всіх вихідних ребер з вершини U.
- `h(Вершина n, Цільова_вершина goal_node)`: Це евристична функція. Вона повинна бути реалізована для конкретної задачі. Наприклад, для навігації на двовимірній сітці:

- Манхеттенська відстань: $h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|$. Завжди допустима для руху по сітці (як шахова тура).
- Евклідова відстань: $h(n) = \sqrt{(x_n - x_{goal})^2 + (y_n - y_{goal})^2}$. Завжди допустима для руху по прямій.

– `g_score[n]`: Зберігає фактичну вартість шляху від `start_node` до `n`. Оновлюється під час релаксації.

– `f_score[n]`: Зберігає оцінку загальної вартості шляху від `start_node` через `n` до `goal_node`. Використовується для пріоритету в `open_set`.

– `predecessor[n]`: Дозволяє відновити фактичний шлях, рухаючись від `goal_node` назад до `start_node`.

– `ЯКЩО current_node == goal_node`: Умова завершення. Коли цільова вершина вилучається з `open_set`, це означає, що до неї знайдено найкоротший шлях, оскільки A* завжди обирає найперспективнішу вершину.

– `ЯКЩО МНОЖИНА_МІСТИТЬ(closed_set, current_node)`: Запобігає повторній обробці вже повністю оброблених вершин, покращуючи ефективність.

– `g_tentative < g_score[neighbor]`: Умова релаксації, подібна до Дейкстри. Якщо знайдено коротший шлях до сусіда, оновлюємо його `g_score`, `f_score` та `predecessor`.

– `ДОДАТИ_В_ЧЕРГУ_З_ПРІОРИТЕТОМ(open_set, (f_score[neighbor], neighbor))`: Додаємо сусіда до `open_set` з оновленою оцінкою. Якщо він вже був у черзі, його пріоритет буде оновлений (або додано новий запис, а старий ігнорується при вилученні).

1.5. Часова та просторова складність

Часова складність: Часова складність алгоритму A* залежить від кількох факторів:

1. Якість евристичної функції: Чим краща (точніша, але все ще допустима) евристика, тим менше вершин буде розширено.

2. Тип графа (щільний/розріджений).
3. Реалізація черги з пріоритетом.

– У найгіршому випадку: Якщо евристика неінформативна або взагалі відсутня ($h(n) = 0$), A* деградує до алгоритму Дейкстри. Якщо ж граф необмежений, і евристика погана, складність може бути експоненціальною $O(b^d)$, де b — коефіцієнт розгалуження, а d — глибина рішення.

– У типових випадках (з гарною евристикою): Продуктивність може бути значно кращою. Якщо евристика узгоджена і граф не дуже "складний", A* може працювати за час, близький до $O(E \log V)$ або $O(E)$, подібно до Дейкстри на розріджених графах.

Просторова складність: Просторова складність A* становить $O(V + E)$ (для зберігання графа) плюс $O(V)$ для масивів `g_score`, `f_score`, `predecessor`, `open_set` та `closed_set`. У найгіршому випадку (коли доводиться досліджувати значну частину графа), `open_set` та `closed_set` можуть зберігати до $O(V)$ вершин. На деяких типах задач, де простір стану величезний, A* може бути обмежений пам'яттю, якщо йому доводиться зберігати багато вершин у `open_set`.

2. Область застосування

Алгоритм A* є надзвичайно універсальним і широко застосовується у багатьох галузях, де потрібен ефективний пошук оптимальних шляхів у великих просторах станів:

1. Штучний інтелект:
    - Ігровий AI: Пошук шляхів для персонажів у відеоіграх (від стратегій до RPG), де карти можуть бути великими, а персонажі повинні знаходити найкоротші маршрути до цілей, уникаючи перешкод.
    - Планування: У системах планування, де агенту потрібно знайти послідовність дій для досягнення мети (наприклад, планування маршрутів для роботів, логістичні операції).
    - Вирішення головоломок: Такі ігри, як "П'ятнашки" (15-puzzle) або "Кубик Рубіка", можуть бути змодельовані як графи станів, де A* може знайти найкоротшу послідовність ходів.

2. Робототехніка:
    - Навігація роботів: Планування руху роботів у фізичному середовищі, де потрібно уникати перешкод та знаходити найефективніший шлях.
    - Автономні транспортні засоби: Хоча вони використовують складніші системи, базові принципи A* лежать в їхній основі.
3. Картографія та навігація:
    - GPS-навігатори: Хоча Google Maps та інші системи використовують більш складні та розподілені алгоритми, основна ідея використання евристик для прискорення пошуку шляху бере початок від A*.
4. Мережева маршрутизація:
    - У деяких випадках, де вартість переходу між вузлами мережі відома, A* може бути використаний для знаходження оптимальних маршрутів.
5. Біоінформатика:
    - Пошук шляхів у мережах взаємодії білків або для вирішення проблем згортання білків.
6. Компілятори:
    - Для оптимізації коду, де пошук найефективнішої послідовності інструкцій може бути змодельований як задача пошуку шляху.

Приклади застосування

– Пошук шляху для персонажа в комп’ютерній грі: Уявіть, що персонаж гравця знаходиться на великій карті, де є стіни, річки та інші перешкоди. Коли гравець клікає на цільову точку, AI персонажа має знайти найкоротший шлях до неї, обходячи перешкоди. Карта може бути змодельована як сітка (граф), де кожна клітинка — це вершина, а перехід між сусідніми клітинками — ребро з вагою (наприклад, 1, якщо клітинка вільна, або нескінченність, якщо це перешкода). Евристичною функцією $h(n)$ може бути пряма (евклідова або манхеттенська) відстань від поточної клітинки до цільової. A*

ефективно знайде найкоротший шлях, розширюючи лише ті клітинки, які здаються “ближчими” до мети.

- Робот-кур’єр на складі: На великому складі робот-кур’єр повинен доставити товар з однієї точки в іншу, уникаючи завалених проходів та інших роботів. Склад може бути представлений як граф з проходами (ребра) та перехрестями (вершини). Ваги ребер можуть відображати час проїзду або ризик зіткнення. A* з евристикою, що базується на географічній відстані, може спланувати оптимальний маршрут, мінімізуючи як шлях, так і ризики, прискорюючи доставку.
- Розв’язання головоломки “П’ятнашки”: У цій головоломці потрібно переміщати плитки по сітці, щоб розташувати їх у певному порядку. Кожен стан дошки є вершиною графа, а переміщення плитки — ребром. Мета — знайти найкоротшу послідовність ходів до виграшного стану. Евристикою $h(n)$ може бути кількість плиток, які знаходяться не на своєму місці, або сума манхеттенських відстаней кожної плитки до її цільової позиції. A* ефективно знаходить рішення, мінімізуючи кількість ходів.

Переваги та Недоліки

Переваги алгоритму A*:

- Оптимальність: Завдяки використанню допустимої евристики A* гарантовано знаходить найкоротший (оптимальний) шлях, якщо такий існує.
- Повнота: Гарантовано знайде шлях, якщо він існує.
- Ефективність: Використовуючи евристику, A* значно звужує простір пошуку порівняно з неінформованими алгоритмами (такими як BFS або Дейкстра, які розширюють “сліпо”). Він “цілеспрямовано” рухається до мети.
- Універсальність: Застосовний до широкого кола задач, що моделюються графами, де є чітка ціль і можна визначити евристику.
- Гнучкість: Якість рішення (оптимальність проти швидкості) можна регулювати, змінюючи допустимість евристики.

Недоліки алгоритму A*:

– Пам’ять: У найгіршому випадку A* може займати багато пам’яті, оскільки йому потрібно зберігати всі розширені (досліджені) вершини у `open_set` та `closed_set`. Для великих просторів станів це може призвести до переповнення пам’яті.

– Якість евристики: Продуктивність A* сильно залежить від якості (точності та допустимості/узгодженості) евристичної функції. Погана евристика може змусити A* деградувати до Дейкстри або навіть гірше.

– Не працює з від’ємними вагами ребер: Як і Дейкстра, A* не може коректно працювати з графами, які містять від’ємні ваги ребер, якщо тільки евристика не є досконалою або не вживаються спеціальні заходи.

– Пошук лише одного шляху: Знаходить найкоротший шлях лише до однієї цільової вершини. Якщо потрібно знайти шлях до кількох цілей, доведеться запускати його кілька разів.

Контрольні запитання

1. Визначення A*: *Що таке алгоритм A*? Чим він відрізняється від неінформованих алгоритмів пошуку?
2. Функція $f(n)$: Як обчислюється функція оцінки $f(n)$? Що означають компоненти $g(n)$ та $h(n)$?
3. Допустимість: Що означає, що евристична функція $h(n)$ є допустимою? Який наслідок це має для оптимальності алгоритму A*?
4. Узгодженість: Що означає, що евристична функція $h(n)$ є узгодженою? Чому узгоджена евристика завжди є допустимою?
5. Черга з пріоритетом: Яка структура даних є ключовою для ефективної роботи A*, і за яким критерієм вона пріоритезує вершини?
6. Відкритий/Закритий список: Для чого використовуються “відкритий список” (`open_set`) та “закритий список” (`closed_set`) в алгоритмі A*?

7. Покроковий опис: Опишіть основний цикл алгоритму A* після ініціалізації, включаючи вибір вершини, перевірку мети та обробку сусідів.

8. Обмеження на ваги: Чи може алгоритм A* працювати з від'ємними вагами ребер? Поясніть.

9. Часова складність: Від яких факторів залежить часова складність алгоритму A*? Яка його складність у найгіршому випадку?

10. Застосування: Наведіть два приклади реальних задач, де алгоритм A* є корисним.

Методичні вказівки до лабораторної роботи

1. Назва

Лабораторна робота: Реалізація Евристичного Пошуку A* на Сітці

2. Мета

Метою даної лабораторної роботи є набуття практичних навичок у реалізації алгоритму A* для пошуку найкоротших шляхів у графі, що моделюється як сітка, розуміння ролі евристичної функції та її властивостей.

3. Завдання

a) Моделювання сітки як графа: Реалізувати представлення сітки (наприклад, $N \times M$ клітинок, де деякі є перешкодами) як зваженого графа (списками суміжності або неявно).

b) Реалізація евристичних функцій: Реалізувати Манхеттенську та Евклідову відстані як евристичні функції `h(n)`.

c) Реалізація черги з пріоритетом: Створити власну просту реалізацію черги з пріоритетом.

d) Реалізація алгоритму A*: Розробити функцію **`aStar(Graph G, int start_node, int goal_node, Function h)`**, яка повертає масив попередників для відновлення шляху.

e) Відновлення шляху: Реалізувати функцію **`reconstructPath(int[] predecessor, int start_node, int target_node)`**, яка відновлює та виводить найкоротший шлях.

f) Тестування: Протестувати реалізований алгоритм на різних сітках:

- Сітка без перешкод.
- Сітка з перешкодами.
- Сітка, де шлях не існує.
- Порівняти поведінку A* з різними евристиками.

4. Опис

Вам необхідно розробити програмний додаток на обраній мові програмування. Додаток повинен дозволяти:

– Створювати сітки, задаючи їхній розмір та розташування перешкод.

– Перетворювати клітинки сітки на вершини графа.

– Викликати функцію **`aStar`** для знаходження найкоротшого шляху між заданими початковою та цільовою клітинками.

– Виводити знайдений шлях, його вартість, а також, можливо, візуалізувати розширені вузли.

Використання списків суміжності або неявної генерації сусідів для представлення графа є обов'язковим.

5. Порядок Виконання

a) Підготовка:

- Створіть новий проект.
- Визначте, як будете представляти сітку (наприклад, 2D масив символів).
- Визначте, як будете перетворювати координати сітки на індекси вершин графа і навпаки.

b) Реалізація евристик:

- Напишіть функції `manhattan_distance(node1_coords, node2_coords)` та `euclidean_distance(node1_coords, node2_coords)`.

c) Реалізація черги з пріоритетом:

- Якщо ваша мова програмування не має вбудованої черги з пріоритетом, реалізуйте її (наприклад, на основі `heapq` в Python).

d) Реалізація алгоритму A*:

- Створіть функцію `aStar(grid, start_coords, goal_coords, heuristic_function)`.
- Ініціалізуйте `g_score`, `f_score`, `predecessor`, `open_set`, `closed_set`.
- Реалізуйте основний цикл A*, включаючи вилучення мінімального елемента, перевірку мети, додавання до `closed_set` та релаксацію сусідів.

e) Реалізація відновлення шляху:

- Створіть функцію `reconstructPath(predecessor, start_coords, goal_coords)`.

f) Тестування:

- Створіть кілька тестових сіток різного розміру та складності (з перешкодами, без перешкод, з неможливими шляхами).
- Для кожної сітки та пари (старт, ціль):
  - Запустіть A* з Манхеттенською евристикою. Виведіть шлях та його вартість.
  - Запустіть A* з Евклідовою евристикою. Виведіть шлях та його вартість.
  - (Опціонально) Візуалізуйте шлях на сітці.

5. Результат

Після виконання всіх завдань програми повинні виводити:

– Для кожного тестового сценарію та евристики:

- Повідомлення, чи знайдено шлях.
- Якщо шлях знайдено: послідовність клітинок (координат) у найкоротшому шляху та його загальну вартість.

- Якщо шлях не знайдено: відповідне повідомлення.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де:
    - Реалізовано моделювання сітки як графа (неявно або явно).
    - Надано функціональні реалізації Манхеттенської та Евклідової евристик.
    - Надано функціональну реалізацію алгоритму A*.
    - Надано функцію для відновлення найкоротшого шляху.
    - Код має бути добре структурованим, читабельним та містити необхідні коментарі, що пояснюють ключові частини реалізації.
2. Продемонструвати роботу додатку:
    - Продемонструвати роботу алгоритму A* на 3-4 прикладах сіток різної складності, включаючи випадки з перешкодами та неможливими шляхами.
    - Показати, як різні евристики впливають на шлях та ефективність (наприклад, порівняти кількість розширених вузлів для Манхеттенської та Евклідової відстаней).
    - Показати коректність виводу знайденого шляху та його вартості.
3. Бути готовим відповісти на питання:
    - Принцип роботи: Детально пояснити покрокову роботу алгоритму A*, включаючи ініціалізацію, роль `f(n)`, `g(n)`, `h(n)`, використання відкритого/закритого списків.
    - Евристики: Пояснити поняття допустимості та узгодженості евристики. Навести приклади допустимих та недопустимих евристик для задачі пошуку шляху на сітці.
    - Оптимальність та повнота: Пояснити, за яких умов A* є оптимальним та повним.

- Складність: Обґрунтувати часову та просторову складність алгоритму A*. Від яких факторів вона залежить?
- Порівняння: Порівняти A* з BFS та Дейкстрою, вказавши їхні переваги та недоліки.
- Застосування: Навести 2-3 приклади реальних задач, де алгоритм A* є незамінним.

Тести

1. Що таке $f(n)$ у алгоритмі A*?

   А) Фактична вартість від старту до n.

   Б) Оцінка вартості від n до мети.

   В) Сума фактичної вартості від старту до n та оцінки вартості від n до мети.

   Г) Вага ребра від n до сусіда.

2. Що означає, що евристична функція $h(n)$ є допустимою?

   А) Вона завжди дорівнює 0.

   Б) Вона ніколи не переоцінює фактичну вартість до мети.

   В) Вона завжди точно оцінює вартість до мети.

   Г) Вона завжди недооцінює вартість до мети.

3. Яка структура даних є ключовою для алгоритму A* для вибору наступної вершини для розширення?

   А) Стек.

   Б) Звичайна черга (FIFO).

   В) Черга з пріоритетом.

   Г) Хеш-таблиця.

4. Якщо $h(n) = 0$ для всіх n, алгоритм A* поводиться як:

   А) Алгоритм Беллмана-Форда.

   Б) Алгоритм Дейкстри.

В) Жадібний пошук.

Г) DFS.

5. Який список (множина) використовується для зберігання вершин, які вже були повністю оброблені A*?

   А) Відкритий список.

   Б) Закритий список.

   В) Черга з пріоритетом.

   Г) Список суміжності.

Відповіді: 1 - В), 2 - Б), 3 - В), 4 - Б),5 - Б).

Тести підвищеної складності

1. Якщо евристична функція A* є узгодженою, то вона також:

   А) Недопустима.

   Б) Допустима.

   В) Неінформативна.

   Г) Немонотонна.

2. Чому A* може бути обмежений пам’яттю на великих графах, навіть якщо час виконання є прийнятним?

   А) Через високу рекурсію.

   Б) Через необхідність зберігати всі розширені вузли у відкритому та закритому списках.

   В) Через експоненціальне зростання кількості ребер.

   Г) Через необхідність зберігати транспонований граф.

3. Яка з цих евристик є допустимою для пошуку найкоротшого шляху на сітці, де рух можливий лише по горизонталі та вертикалі (як у шахах, тура)?

   А) Евклідова відстань.

   Б) Манхеттенська відстань.

В) Вартість найдовшого шляху.

Г) Кількість вершин у графі.

4. A* не може коректно працювати з графами, що містять:

   А) Цикли.

   Б) Кілька компонент зв’язності.

   В) Ребра з від’ємними вагами.

   Г) Нульові ваги ребер.

5. Що відбудеться, якщо евристична функція $h(n)$ переоцінює фактичну вартість до мети (тобто, вона недопустима)?

   А) Алгоритм A* завжди знайде оптимальний шлях.

   Б) Алгоритм A* ніколи не знайде шлях.

   В) Алгоритм A* може не знайти оптимальний шлях, але буде швидшим.

   Г) Алгоритм A* буде деградувати до BFS.

6. Яка з цих ситуацій є типовим застосуванням алгоритму A*?

   А) Виявлення циклів у фінансових мережах.

   Б) Планування маршруту для робота, що орієнтується в складному, невідомому середовищі.

   В) Обчислення найкоротшого шляху в Інтернеті між двома комп’ютерами.

   Г) Визначення порядку виконання завдань з залежностями.

7. Чому евристична функція є важливою для A*?

   А) Вона забезпечує оптимальність шляху.

   Б) Вона дозволяє алгоритму використовувати менше пам’яті.

   В) Вона спрямовує пошук до мети, роблячи його ефективнішим.

   Г) Вона дозволяє обробляти від’ємні ваги ребер.

8. Якщо A* знаходить коротший шлях до вершини, яка вже знаходиться у відкритому списку, що він робить?

   А) Ігнорує новий шлях.

   Б) Додає цю вершину до закритого списку.

   В) Оновлює вартість і пріоритет цієї вершини у відкритому списку.

   Г) Перериває роботу з помилкою.

9. Яке твердження правильне щодо зв'язку між допустимістю та узгодженістю евристик?

   А) Узгоджена евристика є допустимою, але допустима не завжди є узгодженою.

   Б) Допустима евристика є узгодженою, але узгоджена не завжди є допустимою.

   В) Вони не пов'язані між собою.

   Г) Обидві умови означають одне й те ж саме.

10. У контексті ігрового AI, що є найбільш типовим для евристичної функції $h(n)$?

    А) Час, необхідний для обробки всіх сусідів.

    Б) Фактична відстань від гравця до мети.

    В) Оцінка відстані від поточної позиції персонажа до цільової, зазвичай пряма лінія або манхеттенська.

    Г) Кількість доступних ресурсів для персонажа.

Відповіді:

1 Б). Допустима.

2 Б). Через необхідність зберігати всі розширені вузли у відкритому та закритому списках.

3 Б). Манхеттенська відстань.

4 В). Ребра з від'ємними вагами.

5 В). Алгоритм A* може не знайти оптимальний шлях, але буде швидшим.

6 Б). Планування маршруту для робота, що орієнтується в складному, невідомому середовищі.

7 В). Вона спрямовує пошук до мети, роблячи його ефективнішим.

8 В). Оновлює вартість і пріоритет цієї вершини у відкритому списку.

9 А). Узгоджена евристика є допустимою, але допустима не завжди є узгодженою.

10 В). Оцінка відстані від поточної позиції персонажа до цільової, зазвичай пряма лінія або манхеттенська.

Завдання для самостійної роботи

1. Реалізація A* на сітці: Реалізуй алгоритм A* для пошуку найкоротшого шляху на сітці (2D масив), де деякі клітинки є перешкодами. Використай Манхеттенську та Евклідову відстані як евристики.

2. Візуалізація A*: Додай візуалізацію до своєї реалізації A*: показуй шлях, розширені вузли (відкритий та закритий списки) різними кольорами на кожному кроці.

3. Порівняння евристик: Запусти A* на кількох тестових сітках з різними евристиками (Манхеттенська, Евклідова, а також $h(n) = 0$ - що деградує до Дейкстри) та порівняй кількість розширених вузлів та час виконання. Проаналізуй, як якість евристики впливає на ефективність.

4. Адаптація для пошуку найдовшого шляху: Досліди, як A* може бути адаптований для пошуку найдовшого шляху в орієнтованому ациклічному графі (DAG) з позитивними вагами (або найкоротшого шляху з негативними вагами).

5. Зважений A*: Досліди та реалізуй “ Зважений A*”, де функція оцінки $f(n) = g(n) + \epsilon \cdot h(n)$, де ϵ 1. Як це впливає на оптимальність та швидкість?

6. Імплементація IDA*: Досліди та спробуй реалізувати Iterative Deepening A* (IDA*), який є варіацією A*, що оптимізує використання пам’яті.

7. Пошук шляху на карті: Знайди невеликий набір реальних географічних даних (наприклад, перехрестя та дороги з OpenStreetMap) та застосуй A* для пошуку найкоротшого маршруту.
8. Розв'язання головоломки "П'ятнашки": Змоделюй головоломку "П'ятнашки" як задачу пошуку шляху в графі. Реалізуй A* для її розв'язання, використовуючи, наприклад, "кількість плиток не на своєму місці" або "суму манхеттенських відстаней" як евристику.
9. A* для багатокористувацьких ігор: Обміркуй, як A* можна адаптувати для багатокористувацьких ігор, де перешкоди можуть бути динамічними (рухомими).
10. Обробка негативних ваг (якщо це можливо): Досліди, як A* може бути використаний у графах з негативними вагами, але без негативних циклів. Це вимагатиме попереднього перетворення графа (наприклад, за допомогою Джонсона) або адаптації алгоритму.

Мінімальні кістякові дерева (MST): алгоритми Крускала, Прима, Борувки.

У попередніх розділах ми докладно вивчили, як графи моделюють зв'язки між об'єктами, а також як знаходити найкоротші шляхи між ними. Тепер ми звернемося до іншої фундаментальної задачі в теорії графів, яка має величезне значення для проектування мереж та оптимізації витрат: побудова мінімальних кістякових дерев.

Уявіть, що тобі потрібно з'єднати кілька міст телекомунікаційними кабелями, або прокласти водопровідні труби між будинками, або спроектувати мережу доріг, щоб усі населені пункти були доступні. Мета — зробити це таким чином, щоб загальна вартість будівництва (довжина кабелів, труб, доріг) була мінімальною, але при цьому всі точки були зв'язані. Саме цю проблему вирішують мінімальні кістякові дерева.

Кістякове дерево (або охоплююче дерево, spanning tree) зв'язного неорієнтованого графа — це підграф, який є деревом (тобто не містить циклів) і при цьому з'єднує всі вершини оригінального графа. Якщо граф зважений (тобто, кожне ребро має певну вагу, що може представляти вартість, відстань, час тощо), то мінімальне кістякове дерево (MST) — це кістякове дерево, сума ваг усіх ребер якого є мінімально можливою.

Знаходження MST є класичною задачею оптимізації, яка має численні застосування в реальному світі. Це дозволяє нам ефективно будувати мережі, мінімізуючи витрати, забезпечуючи при цьому повну зв'язність. У цьому підрозділі ми детально розглянемо два найвідоміші та найефективніші алгоритми для побудови мінімальних кістякових дерев: алгоритм Прима, Борувки та алгоритм Крускала.

Алгоритм Прима для пошуку мінімального кістякового дерева

Ласкаво просимо до розділу, присвяченого одному з фундаментальних "жадібних" алгоритмів — алгоритму Прима. У попередніх розділах ми розглянули базові концепції графів, їхні види та способи представлення у пам'яті. Тепер ми заглибимося у практичну задачу: як з'єднати всі вершини графа з мінімальними витратами. Ця задача є ключовою у багатьох галузях, від проектування комп'ютерних мереж до логістики.

1.1. Основні визначення

Перш ніж перейти до алгоритму, давайте формалізуємо поняття.

- Зв'язний неорієнтований зважений граф: Це граф $G = (V, E, w)$, де:
  - V — множина вершин (наприклад, міста).
  - E — множина ребер, що з'єднують пари вершин. Граф неорієнтований, тому ребро (u, v) ідентичне ребру (v, u). Граф зв'язний, тобто існує шлях між будь-якими двома вершинами.
  - w — функція ваги, що кожному ребру $e \in E$ ставить у відповідність деяке числове значення $w(e)$ (вагу, вартість, відстань).
- Кістякове дерево: Це підграф $T = (V, E')$ вихідного графа G, який є деревом і включає *усі* вершини V з вихідного графа. Оскільки це дерево, воно не містить циклів і є зв'язним. Для графа з $|V|$ вершинами будь-яке кістякове дерево буде містити рівно $|V| - 1$ ребер.
- Мінімальне кістякове дерево (МКД, MST): Це кістякове дерево, сума ваг ребер якого є мінімально можливою серед усіх можливих кістякових дерев цього графа. Формально, якщо $T = (V, E')$ є кістяковим деревом, його вага $W(T) = \sum_{e \in E'} w(e)$. МКД — це таке

дерево T_{MST}, що $W(T_{MST}) \leq W(T)$ для будь-якого іншого кістякового дерева T.

Важливо зазначити, що граф може мати декілька різних МКД, якщо в ньому є ребра з однаковою вагою. Проте вага всіх МКД для одного графа завжди буде однаковою.

1.2. Ідея алгоритму Прима

Алгоритм Прима — це класичний приклад жадібного алгоритму. Його стратегія полягає в тому, щоб на кожному кроці робити локально оптимальний вибір, сподіваючись, що це призведе до глобально оптимального рішення. У випадку з МКД ця стратегія виявляється правильною.

Ідея алгоритму дуже інтуїтивна. Замість того, щоб розглядати всі ребра графа одразу, ми "вирощуємо" наше мінімальне кістякове дерево з однієї довільно обраної початкової вершини.

1. Починаємо з дерева, що складається з однієї довільної вершини.
2. На кожному наступному кроці ми знаходимо найлегше (найдешевше) ребро, яке з'єднує вершину, що *вже є* в нашому дереві, з вершиною, якої в дереві *ще немає*.
3. Додаємо це ребро та нову вершину до нашого дерева.
4. Повторюємо крок 2 і 3, доки всі вершини вихідного графа не будуть включені до дерева.

Оскільки на кожному кроці ми додаємо одну вершину, для графа з $|V|$ вершинами ми виконаємо $|V| - 1$ таких кроків.

1.3. Теоретичне обґрунтування (Властивість розрізу)

Гарантія, що ми знаходимо саме мінімальне кістякове дерево криється у фундаментальній властивості МКД, відомій як властивість розрізу.

- Розріз: Розріз графа $G = (V, E)$ — це поділ множини вершин V на дві довільні непорожні підмножини, що не перетинаються, — S і $V - S$.
- Ребро, що перетинає розріз: Це ребро (u, v), де одна вершина належить S, а інша — $V - S$.

Теорема про безпечне ребро: Нехай $(S, V - S)$ — довільний розріз графа. Якщо ребро $e = (u, v)$ є ребром з мінімальною вагою серед усіх ребер, що

перетинають цей розріз, то це ребро e належить принаймні одному МКД цього графа.

Алгоритм Прима на кожному кроці, по суті, використовує цю теорему. Множина S — це вершини, вже включені в наше дерево. Множина $V - S$ — це всі інші вершини. Алгоритм знаходить найлегше ребро, що перетинає розріз $(S, V - S)$, і додає його до дерева. Оскільки цей вибір є "безпечним" згідно з теоремою, він не завадить нам у підсумку побудувати МКД.

1.4. Реалізація та структури даних

Ефективність алгоритму Прима сильно залежить від того, як ми реалізуємо крок "знайти найлегше ребро, що з'єднує вершину з дерева з вершиною поза деревом".

Для реалізації нам знадобляться такі структури даних:

- `dist[v]`: Масив для зберігання ваги найлегшого ребра, що з'єднує вершину `v` (яка ще не в дереві) з будь-якою вершиною, що вже є в дереві. Початково `dist[v]` для всіх вершин дорівнює нескінченності (∞), крім стартової, для якої `dist` дорівнює 0.
- `parent[v]`: Масив для зберігання "батьківської" вершини для `v` у створюваному дереві. Це дозволить нам відновити саме дерево після завершення алгоритму.
- `inMST[v]`: Булевий масив (або множина), що позначає, чи включена вершина `v` вже до МКД.

Наївний підхід (складність $O(V^2)$)

Найпростіший спосіб — на кожному кроці лінійно проходити по всіх вершинах, що ще не в дереві, і знаходити ту, для якої значення в `dist` є мінімальним.

Псевдокод:

```
function PrimMST_Naive(graph G = (V, E)):
  // Ініціалізація
  for each vertex v in V:
    dist[v] = ∞
    parent[v] = NULL
    inMST[v] = false

  // Вибираємо довільну стартову вершину s
  dist[s] = 0
```

```
  // Головний цикл виконується |V| разів
  for i from 1 to |V|:
    // 1. Знайти вершину u, яка ще не в МКД (inMST[u] == false)
    //    і має мінімальне значення dist[u]
    u = find_min_dist_vertex(dist, inMST)

    // Якщо u не знайдено (для незв'язних графів), зупиняємось
    if u == NULL:
      break

    // 2. Додати вершину u до МКД
    inMST[u] = true

    // 3. Оновити значення dist для всіх сусідів v вершини u
    for each neighbor v of u:
      // w(u, v) - вага ребра між u та v
      if inMST[v] == false and w(u, v) < dist[v]:
        dist[v] = w(u, v)
        parent[v] = u

  // Результатом є набір ребер (parent[v], v) для всіх v, крім старто
вої
  return parent_array_to_MST(parent)
```

Цей підхід є ефективним для щільних графів, де кількість ребер $|E|$ наближається до максимально можливої, тобто $O(|V|^2)$. Пошук вершини `u` на кожному кроці займає $O(|V|)$ часу, і цей крок повторюється $|V|$ разів, що дає загальну складність $O(|V|^2)$.

Ефективний підхід з використанням черги з пріоритетом (складність $O(E \log V)$)

Пошук вершини з мінімальною відстанню на кожному кроці можна значно прискорити, якщо використати чергу з пріоритетом. У цій черзі ми будемо зберігати вершини, які ще не ввійшли до МКД. Пріоритетом кожної вершини `v` буде значення `dist[v]`.

Алгоритм з чергою з пріоритетом

1. Ініціалізуємо `dist` для всіх вершин як ∞, а для стартової `s` як 0.
2. Створюємо чергу з пріоритетом `PQ` і додаємо до неї всі вершини. Пріоритет вершини `v` визначається значенням `dist[v]`.

3. Поки `PQ` не порожня:

    i. Вилучаємо з `PQ` вершину `u` з найменшим пріоритетом (найменшою відстанню). Ця операція називається `extractMin`.

    j. Додаємо `u` до набору вершин МКД.

    k. Для кожного сусіда `v` вершини `u`:

    l. Якщо `v` все ще перебуває в `PQ` і вага ребра `(u, v)` менша за поточну записану відстань до `v` (`dist[v]`), то ми оновлюємо `dist[v]` і `parent[v]`.

    iv. Після оновлення `dist[v]` потрібно також оновити пріоритет `v` у черзі `PQ`. Ця операція називається `decreaseKey`.

Псевдокод такий:

```
function PrimMST_PriorityQueue(graph G = (V, E)):
  // Ініціалізація
  for each vertex v in V:
    dist[v] = ∞
    parent[v] = NULL

  // Вибираємо довільну стартову вершину s
  dist[s] = 0

  // Створюємо чергу з пріоритетом і додаємо всі вершини
  PQ = new PriorityQueue()
  for each vertex v in V:
    PQ.insert(v, dist[v])

  // Головний цикл
  while PQ is not empty:
    // 1. Вилучити вершину u з найменшим dist
    u = PQ.extractMin()

    // 2. Оновити значення dist для сусідів
    for each neighbor v of u:
      // w(u,v) - вага ребра
      // PQ.contains(v) перевіряє, чи v ще не в МКД
      if PQ.contains(v) and w(u, v) < dist[v]:
        parent[v] = u
        dist[v] = w(u, v)
        PQ.decreaseKey(v, dist[v]) // Оновити пріоритет v
```

```
  // Результат - набір ребер (parent[v], v)
  return parent_array_to_MST(parent)
```
Аналіз складності:

- Побудова черги з пріоритетом: $O(|V|)$.
- Цикл `while` виконується $|V|$ разів. Операція `extractMin` для бінарної купи (binary heap) займає $O(\log V)$. Сумарно: $O(|V|\log V)$.
- Внутрішній цикл `for` для сусідів виконується сумарно для всіх вершин $2|E|$ разів (для неорієнтованого графа). Операція `decreaseKey` для бінарної купи також займає $O(\log V)$. Сумарно: $O(|E|\log V)$.

Загальна складність алгоритму становить $O(|V|\log V + |E|\log V)$. Оскільки у зв'язному графі $|E| \geq |V| - 1$, складність спрощується до $O(|E|\log V)$. Цей варіант значно ефективніший для розріджених графів (sparse graphs), де $|E|$ набагато менше за $|V|^2$.

2. Область застосування

Алгоритм Прима та загалом задача пошуку МКД мають широке застосування в реальному світі:

- Проектування мереж: Найбільш очевидне застосування. Проектування комп'ютерних, телекомунікаційних, електричних мереж, а також мереж трубопроводів (вода, газ, нафта) з мінімальними витратами на матеріали.
- Кластерний аналіз: У машинному навчанні та аналізі даних МКД може використовуватися для ідентифікації кластерів. Якщо видалити найдовші ребра з МКД, граф розпадеться на групи точок (кластери).
- Наближені алгоритми для NP-складних задач: МКД є важливим кроком у деяких наближених алгоритмах для розв'язання складніших задач, наприклад, задачі комівояжера. Вага МКД є нижньою межею для довжини найкоротшого гамільтонового циклу.
- Дизайн мікросхем: Для з'єднання контактів на друкованій платі або кристалі з мінімальною довжиною провідників.
- Аналіз зображень: Для сегментації зображень, де вершини — це пікселі або регіони, а вага ребер — різниця в кольорі чи текстурі.

3. Приклад застосування

Розглянемо невеликий зважений граф і покроково застосуємо до нього алгоритм Прима.

Вхідний граф: Нехай у нас є граф з вершинами {A, B, C, D, E, F}. Ребра та їх ваги:

- (A, B) - 2
- (A, C) - 3
- (A, D) - 3
- (B, C) - 4
- (B, E) - 3
- (C, D) - 5
- (C, F) - 6
- (D, F) - 7
- (E, F) - 8

Ми будемо використовувати підхід з чергою з пріоритетом.

Крок 0: Ініціалізація

- Вибираємо стартову вершину, нехай це буде A.
- **`dist`** = {A:0, B:∞, C:∞, D:∞, E:∞, F:∞}
- **`parent`** = {A:null, B:null, C:null, D:null, E:null, F:null}
- **`inMST`** = {}
- МКД = {}

Крок 1:

1. Вилучаємо з черги вершину з найменшим **`dist`** — це A (dist=0).
2. Додаємо A до **`inMST`**. **`inMST`** = {A}.
3. Оновлюємо **`dist`** для сусідів A:
    - B: **`dist[B]`** стає 2 (бо 2 < ∞), **`parent[B]`** = A.
    - C: **`dist[C]`** стає 3 (бо 3 < ∞), **`parent[C]`** = A.

- D: **dist[D]** стає 3 (бо 3 < ∞), **parent[D]** = A.

4. Стан **dist**: {A:0, B:2, C:3, D:3, E:∞, F:∞}.

Крок 2:

5. Вершина з найменшим **dist** серед тих, що не в **inMST** — це B (dist=2). Вилучаємо її.
6. Додаємо ребро (A, B) до МКД. **inMST** = {A, B}. МКД = {(A, B)}.
7. Оновлюємо **dist** для сусідів B (C, E):
   - C: вага (B, C) = 4. Поточний **dist[C]** = 3. Оскільки 4 не менше 3, нічого не змінюємо.
   - E: **dist[E]** стає 3 (бо 3 < ∞), **parent[E]** = B.
8. Стан **dist**: {A:0, B:2, C:3, D:3, E:3, F:∞}.

Крок 3:

9. Вершини з найменшим **dist** (3) — це C, D, E. Припустимо, алгоритм вибирає C. Вилучаємо її.
10. Додаємо ребро (A, C) до МКД. **inMST** = {A, B, C}. МКД = {(A, B), (A, C)}.
11. Оновлюємо **dist** для сусідів C (A, B, D, F):
    - A, B вже в **inMST**, ігноруємо.
    - D: вага (C, D) = 5. Поточний **dist[D]** = 3. 5 не менше 3, не змінюємо.
    - F: **dist[F]** стає 6 (бо 6 < ∞), **parent[F]** = C.
12. Стан **dist**: {A:0, B:2, C:3, D:3, E:3, F:6}.

Крок 4:

13. Вершини з найменшим **dist** (3) — це D, E. Вибираємо D. Вилучаємо її.
14. Додаємо ребро (A, D) до МКД. **inMST** = {A, B, C, D}. МКД = {(A, B), (A, C), (A, D)}.
15. Оновлюємо **dist** для сусідів D (A, C, F):

- A, C вже в **inMST**.
- F: вага (D, F) = 7. Поточний **dist[F]** = 6. 7 не менше 6, не змінюємо.

16. Стан **dist**: {A:0, B:2, C:3, D:3, E:3, F:6}.

Крок 5:

17. Вершина з найменшим **dist** — це E (dist=3). Вилучаємо її.
18. Додаємо ребро (B, E) до МКД. **inMST** = {A, B, C, D, E}. МКД = {(A, B), (A, C), (A, D), (B, E)}.
19. Оновлюємо **dist** для сусідів E (B, F):
    - B вже в **inMST**.
    - F: вага (E, F) = 8. Поточний **dist[F]** = 6. 8 не менше 6, не змінюємо.
20. Стан **dist**: {A:0, B:2, C:3, D:3, E:3, F:6}.

Крок 6:

21. Залишилася остання вершина, що не в **inMST** — F. Вилучаємо її.
22. Її **dist** = 6, **parent** = C. Додаємо ребро (C, F) до МКД. **inMST** = {A, B, C, D, E, F}.
23. Всі вершини в дереві. Алгоритм завершено.

Результат: Мінімальне кістякове дерево складається з ребер: (A, B), (A, C), (A, D), (B, E), (C, F).

Загальна вага = $w(A,B) + w(A,C) + w(A,D) + w(B,E) + w(C,F) = 2 + 3 + 3 + 3 + 6 = 17$.

4. Переваги та недоліки

Переваги

– Гарантована оптимальність: Алгоритм завжди знаходить мінімальне кістякове дерево.

– Простота концепції: Ідея “вирощування” дерева від однієї вершини є інтуїтивно зрозумілою.

– Ефективність на щільних графах: Реалізація з часовою складністю $O(V^2)$ є однією з найшвидших для щільних графів, де $|E|$ близьке до $|V|^2$. Вона не вимагає складних структур даних, як-от черга з пріоритетом.

Недоліки

– Неефективність на розріджених графах (для наївної реалізації): Складність $O(V^2)$ є надто повільною для великих розріджених графів.

– Складніша ефективна реалізація: Щоб досягти складності $O(E\log V)$, необхідно реалізувати чергу з пріоритетом з операцією `decreaseKey`, що є нетривіальним (стандартні бібліотечні черги в багатьох мовах програмування не підтримують цю операцію ефективно).

– Не працює на незв'язних графах: Якщо граф незв'язний, алгоритм побудує МКД для тієї компоненти зв'язності, в якій була обрана стартова вершина, а решту вершин проігнорує. Для таких графів потрібно запускати алгоритм для кожної компоненти зв'язності, щоб знайти мінімальний кістяковий ліс (minimum spanning forest).

Контрольні запитання

1. Що таке мінімальне кістякове дерево і чим воно відрізняється від просто кістякового дерева?
2. Поясніть своїми словами "жадібну" стратегію алгоритму Прима.
3. Що таке "розріз" графа і як "властивість розрізу" обґрунтовує коректність алгоритму Прима?
4. Які структури даних використовуються в ефективній реалізації алгоритму Прима і як вони впливають на його асимптотичну складність?
5. Що станеться, якщо запустити алгоритм Прима на графі, що містить ребра з від'ємною вагою? Чи буде результат коректним?
6. Чим відрізняється поведінка алгоритму Прима на щільних та розріджених графах? В якому випадку яка реалізація (наївна чи з чергою з пріоритетом) буде кращою?

7. Чи може у графа бути декілька різних МКД? Якщо так, то за яких умов?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Прима для пошуку мінімального кістякового дерева у зваженому неорієнтованому графі.

2. Мета

– Закріпити теоретичні знання про алгоритм Прима та його застосування.

– Набути практичних навичок реалізації графових алгоритмів.

– Навчитися використовувати відповідні структури даних (списки суміжності, черга з пріоритетом) для оптимізації алгоритму.

– Дослідити вплив вибору структури даних на продуктивність програми.

3. Завдання

a) Спроектувати та реалізувати клас або структуру для представлення зваженого неорієнтованого графа (рекомендовано використовувати списки суміжності).

b) Реалізувати алгоритм Прима для пошуку МКД у заданому графі. Необхідно реалізувати ефективну версію з використанням черги з пріоритетом ($O(E \log V)$).

c) Передбачити можливість завантаження графа з текстового файлу. Формат файлу: перший рядок — кількість вершин $|V|$ та ребер $|E|$; наступні $|E|$ рядків — опис ребер у форматі `u v w` (вершина_1 вершина_2 вага).

d) Програма повинна виводити на екран ребра, що увійшли до МКД, та сумарну вагу МКД.

e) (Додатково, для високої оцінки) Реалізувати візуалізацію початкового графа та отриманого МКД (можна використовувати будь-яку графічну бібліотеку або вивести у форматі Graphviz).

4. Опис

Лабораторна робота полягає у створенні консольного або віконного додатку, що знаходить мінімальне кістякове дерево для заданого графа за допомогою алгоритму Прима. Граф задається у вхідному файлі. Програма повинна коректно обробляти вхідні дані, будувати внутрішнє представлення графа, застосовувати алгоритм і виводити результат у зрозумілому для користувача вигляді.

5. Порядок виконання

a) Аналіз вимог. Уважно прочитайте завдання та визначте основні функціональні блоки програми: читання файлу, представлення графа, реалізація алгоритму Прима, виведення результатів.

b) Проектування структури даних. Виберіть та спроектуйте класи/структури для графа, вершини, ребра. Оптимальним є представлення графа у вигляді списків суміжності, де для кожної вершини зберігається список її сусідів та ваг ребер до них.

c) Реалізація представлення графа. Напишіть код, що створює граф і дозволяє додавати до нього вершини та ребра. Реалізуйте функцію завантаження графа з файлу.

d) Реалізація черги з пріоритетом. Якщо мова програмування має вбудовану чергу з пріоритетом (наприклад, `priority_queue` в C++, `PriorityQueue` в Java/Python), вивчіть її інтерфейс. Зверніть увагу, що для операції `decreaseKey` може знадобитися власна реалізація або обхідний шлях (наприклад, додавання дублікатів з новим пріоритетом).

e) Реалізація алгоритму Прима. Напишіть функцію, що приймає на вхід граф і повертає МКД (наприклад, у вигляді списку ребер). Дотримуйтесь псевдокоду, наведеного в теоретичній частині.

f) Інтеграція та тестування. З'єднайте всі частини програми. Створіть кілька тестових файлів з графами (включно з прикладом з теоретичної частини), перевірте коректність роботи програми та правильність обчислення ваги МКД.

g) Написання звіту. Оформіть звіт, що містить опис виконаної роботи, структуру програми, результати тестування та висновки.

6. Результат

Результатом роботи є програмний продукт, що демонструє знаходження МКД, та звіт з його описом. Програма повинна успішно знаходити МКД для довільного зв'язного зваженого неорієнтованого графа.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
    - Структура даних для представлення графа.
    - Реалізація алгоритму Прима.
    - Функції для завантаження даних та виведення результату.
    - Код має бути добре прокоментований.
2. Продемонструвати роботу додатку:
    - Запустити програму з тестовим файлом, наданим викладачем.
    - Запустити програму з власним тестовим файлом, що містить не менше 10 вершин і 15 ребер.
    - Показати, що виведені ребра МКД та його вага є правильними.
3. Бути готовим відповісти на питання:
    - Питання з розділу "Контрольні запитання".
    - Яку асимптотичну складність має ваша реалізація і чому?
    - Як можна було б змінити ваш код для реалізації наївного варіанту алгоритму ($O(V^2)$)?
    - Що станеться, якщо у вхідному файлі будуть ребра з однаковою вагою? Як це вплине на результат?
    - Як ваша програма поводитиметься, якщо вхідний граф незв'язний? Як можна виправити таку поведінку?

Тести

1. Який основний принцип лежить в основі алгоритму Прима?
    1. Розділяй та володарюй

2. Динамічне програмування
3. Жадібний підхід
4. Повернення з рекурсії

2. Яку часову складність має алгоритм Прима при реалізації на щільному графі з використанням матриці суміжності та пошуку мінімуму в масиві?
    1. $O(E + V\log V)$
    2. $O(V^2)$
    3. $O(E\log V)$
    4. $O(V\log V)$
3. Що робить алгоритм Прима на кожному кроці?
    1. Додає найлегше ребро з усього графа, яке ще не було додано.
    2. Додає найлегше ребро, що з'єднує дві вершини, які ще не в МКД.
    3. Додає найлегше ребро, що з'єднує вершину в МКД з вершиною поза МКД.
    4. Додає довільне ребро, яке не утворює циклу.
4. Що гарантує "властивість розрізу" (cut property)?
    1. Що будь-який розріз графа містить принаймні одне ребро з МКД.
    2. Що найважче ребро розрізу ніколи не потрапить до МКД.
    3. Що найлегше ребро, яке перетинає будь-який розріз, є безпечним для включення в МКД.
    4. Що граф розпадеться на дві частини, якщо видалити ребро з МКД.
5. Для якого типу графів реалізація алгоритму Прима з чергою з пріоритетом ($O(E\log V)$) є значно ефективнішою за наївну ($O(V^2)$)?
    1. Для щільних графів ($|E| \approx |V|^2$)

2. Для розріджених графів ($|E| \approx |V|$)
3. Для повних графів
4. Для дводольних графів

Відповіді: 1. – 3), 2. – 2), 3. – 3), 4. – 3), 5. – 2).

Тести підвищеної складності

1. Що станеться, якщо в графі є ребра з від'ємною вагою?
    1. Алгоритм Прима зациклиться.
    2. Алгоритм Прима видасть помилку.
    3. Алгоритм Прима знайде неправильне МКД.
    4. Алгоритм Прима працюватиме коректно і знайде правильне МКД.
2. Якщо для реалізації черги з пріоритетом в алгоритмі Прима використати Фібоначчієву купу (Fibonaccі heap), якою буде асимптотична складність?
    1. $O(E\log V)$
    2. $O(V\log V + E)$
    3. $O(E\log E)$
    4. $O(V^2)$
3. Розглянемо граф, де всі ребра мають унікальні ваги. Скільки МКД існує для такого графа?
    1. Завжди одне.
    2. Залежить від кількості вершин.
    3. Може бути декілька.
    4. Не можна визначити.
4. Яка структура даних є "вузьким місцем" у реалізації Прима зі складністю $O(E\log V)$?

    1. Представлення графа (списки суміжності).
    2. Черга з пріоритетом, а саме операції `extractMin` та `decreaseKey`.
    3. Масив `parent` для зберігання дерева.
    4. Масив `dist` для зберігання відстаней.
5. Чим принципово відрізняється стратегія алгоритму Прима від стратегії алгоритму Крускала?
    1. Прим будує одне дерево, а Крускал — ліс, який згодом об’єднується.
    2. Прим працює тільки на щільних графах, а Крускал — на розріджених.
    3. Прим жадібний, а Крускал — ні.
    4. Немає принципової різниці, обидва роблять одне й те саме.
6. Щоб знайти максимальне кістякове дерево (дерево з максимальною сумарною вагою ребер), як потрібно модифікувати алгоритм Прима?
    1. Алгоритм не можна модифікувати для цієї задачі.
    2. На кожному кроці вибирати найважче ребро замість найлегшого.
    3. Помножити всі ваги ребер на -1 і запустити звичайний алгоритм Прима.
    4. Правильні відповіді 2 і 3.
7. У наївній реалізації ($O(V^2)$) на кожній ітерації головного циклу виконується пошук вершини `u` з мінімальним `dist[u]`. Скільки часу займає цей крок?
    1. $O(\log V)$
    2. $O(1)$
    3. $O(V)$
    4. $O(E)$

8. Якщо граф незв’язний, що поверне стандартна реалізація алгоритму Прима?

    1. Мінімальний кістяковий ліс.
    2. Помилку виконання.
    3. Мінімальне кістякове дерево для найбільшої компоненти зв’язності.
    4. Мінімальне кістякове дерево для компоненти зв’язності, що містить стартову вершину.

9. Операція `decreaseKey` є критичною для ефективної реалізації Прима. Чому її немає у стандартних реалізаціях багатьох черг з пріоритетом (напр., `std::priority_queue` у C++)?

    1. Тому що вона не потрібна для більшості задач.
    2. Тому що її ефективна реалізація ($O(\log n)$) вимагає доступу до внутрішніх елементів купи, що порушує абстракцію.
    3. Тому що її можна легко зімітувати операціями `pop` та `push`.
    4. Це помилка у дизайні стандартних бібліотек.

10. На якому кроці алгоритм Прима може додати ребро, яке є найважчим у всьому графі?

    1. Ніколи, він завжди вибирає найлегші доступні ребра.
    2. Тільки на останньому кроці.
    3. Це можливо, якщо це ребро є єдиним “мостом” для приєднання ізольованої вершини чи компоненти.
    4. Тільки якщо всі інші ребра вже увійшли до МКД.

Відповіді:

1. 4). Алгоритм Прима коректно працює з ребрами від'ємної ваги, оскільки його логіка базується на порівнянні ваг, а не на їх сумуванні для шляхів (на відміну від деяких алгоритмів пошуку найкоротшого шляху, як-от Дейкстри).

2. 2). З Фібоначчієвою купою операція **`extractMin`** займає амортизований час $O(\log V)$, а **`decreaseKey`** — $O(1)$. Загальна складність стає $|V| \cdot O(\log V) + |E| \cdot O(1) = O(V \log V + E)$.

3. 1). Якщо всі ваги ребер унікальні, то на кожному кроці вибір найлегшого ребра, що перетинає розріз, буде однозначним. Отже, МКД буде єдиним.

4. 2). Представлення графа та масиви є дуже швидкими. Основний час витрачається на операції з чергою з пріоритетом, які виконуються в циклах.

5. 1). Алгоритм Прима "вирощує" один зв'язний компонент (дерево) з початкової вершини. Алгоритм Крускала розглядає ребра по всьому графу від найлегшого до найважчого, об'єднуючи різні компоненти в єдине дерево, тому на проміжних етапах він оперує лісом дерев.

6. 4). Можна або змінити логіку порівняння з `min` на `max`, або інвертувати ваги (помноживши на -1) і шукати "мінімальне" дерево, яке насправді буде максимальним для початкових ваг. Обидва підходи є коректними.

7. 3). Наївний пошук мінімуму в масиві з `N` елементів вимагає лінійного проходу, що займає $O(N)$ часу. В контексті алгоритму кількість вершин у масиві `dist` дорівнює $|V|$, тому складність $O(V)$.

8. 4). Алгоритм досліджує лише ті вершини, які досяжні з початкової. Таким чином, він побудує МКД тільки для тієї компоненти зв'язності, якій належить стартова вершина, і зупиниться, коли всі вершини цієї компоненти будуть оброблені.

9. 2). Стандартні черги з пріоритетом, реалізовані на основі бінарних куп, зазвичай не надають прямого доступу до елементів усередині купи, оскільки це порушило б інкапсуляцію. Для `decreaseKey` потрібно знайти елемент і перемістити його вгору по купі, що вимагає знання його позиції.

10. 3). Це можливо. Якщо граф складається з двох великих компонент, з'єднаних одним-єдиним ребром-"мостом", і це ребро має найбільшу вагу в графі, алгоритм Прима буде змушений вибрати його, щоб з'єднати ці дві частини і включити всі вершини в дерево.

Завдання для самостійної роботи

1. Реалізуйте наївну версію алгоритму Прима ($O(V^2)$) і порівняйте час її виконання з версією на черзі з пріоритетом на кількох графах (розрідженому та щільному).
2. Напишіть програму, яка генерує випадковий зв'язний зважений граф із заданою кількістю вершин та щільністю.
3. Модифікуйте вашу реалізацію алгоритму Прима для знаходження максимального кістякового дерева.
4. Дослідить, що відбувається, коли ви запускаєте алгоритм Прима з різних стартових вершин на графі, де є кілька МКД. Чи завжди результат однаковий? Поясніть чому.
5. Реалізуйте алгоритм для пошуку мінімального кістякового лісу для незв'язного графа.
6. Доведіть математично (або поясніть своїми словами), чому алгоритм Прима не створює циклів.
7. Уявіть, що вартість додавання вершини до мережі не нульова, а залежить від самої вершини. Як би ви модифікували задачу та алгоритм для врахування вартості вершин? Чи буде це ще задача про МКД?
8. Порівняйте алгоритм Прима з алгоритмом Дейкстри для пошуку найкоротших шляхів. Знайдіть спільні риси в їхній структурі та ключові відмінності.
9. Напишіть функцію, яка перевіряє, чи є заданий підграф мінімальним кістяковим деревом для вихідного графа.
10. Проаналізуйте, як вплине на МКД додавання константи C до ваги кожного ребра в графі. Чи залишиться МКД тим самим? А якщо помножити кожну вагу на константу C 0?

Теми лабораторних робіт

1. Порівняльний аналіз продуктивності реалізацій алгоритму Прима ($O(V^2)$ проти $O(E\log V)$).

2. Реалізація алгоритму Прима з візуалізацією процесу побудови МКД крок за кроком.
3. Застосування алгоритму Прима для вирішення задачі кластеризації на 2D-площині.
4. Розробка програми для проектування “найдешевшої” комп’ютерної мережі для офісного центру (граф задається планом будівлі).
5. Порівняння алгоритмів Прима та Крускала: реалізація та аналіз поведінки на різних типах графів.
6. Реалізація алгоритму Прима з використанням Фібоначчієвої купи.
7. Побудова МКД для графа, заданого матрицею відстаней між містами України.
8. Модифікація алгоритму Прима для роботи з графами, де ребра можуть з’являтися та зникати динамічно.
9. Використання МКД для стиснення зображень (розглядаючи пікселі як вершини).
10. Пошук “пляшкового горла” в мережі: знаходження другого за вагою найкращого кістякового дерева.
11. Реалізація паралельної версії алгоритму Прима.
12. Розробка наближеного алгоритму для задачі комівояжера, що використовує МКД як перший крок.

Алгоритм Крускала та структури даних “система неперетинних множин”

У попередньому розділі ми дослідили алгоритм Прима, який “вирощує” мінімальне кістякове дерево (МКД) з однієї вершини, поступово додаючи до нього найближчих сусідів.

Алгоритм Крускала пропонує зовсім іншу, але не менш елегантну “жадібну” стратегію. Замість того, щоб вирощувати одне дерево, він розглядає граф як сукупність окремих “поселень” (вершин) і поступово об’єднує їх найдешевшими “дорогами” (ребрами), доки всі не стануть частиною єдиної держави.

Цей підхід не лише приводить нас до того ж оптимального результату, але й знайомить з надзвичайно потужною та ефективною структурою даних, відомою як система неперетинних множин (DSU).

Мета залишається незмінною: для заданого зв'язного неорієнтованого зваженого графа $G = (V, E, w)$ знайти підграф, що є деревом, охоплює всі вершини V і має мінімально можливу сумарну вагу ребер.

1.1. Ідея алгоритму Крускала

Стратегія Крускала є "глобальною" на відміну від "локальної" стратегії Прима. Замість того, щоб розширюватися від однієї точки, ми розглядаємо всі можливі з'єднання в графі одразу.

1. Створюємо "ліс" T, в якому кожна вершина з V є окремим деревом (що складається з однієї вершини).
2. Сортуємо всі ребра графа E за зростанням їхньої ваги.
3. Послідовно розглядаємо кожне ребро (u, v) з відсортованого списку, від найлегшого до найважчого.
4. Для кожного ребра перевіряємо: чи належать вершини u та v до різних дерев у нашому лісі T?
    - Якщо так, це означає, що додавання цього ребра не створить циклу. Ми додаємо ребро (u, v) до нашого лісу, тим самим об'єднуючи два дерева в одне.
    - Якщо ні, вершини u та v вже знаходяться в одному дереві. Додавання ребра (u, v) створило б цикл, тому ми ігноруємо (відкидаємо) це ребро.
5. Повторюємо крок 4, доки в наше МКД не буде додано $|V| - 1$ ребер. На цьому етапі всі вершини будуть об'єднані в одне дерево, і алгоритм завершується.

Ключове питання, яке виникає: як ефективно перевіряти, чи належать дві вершини до одного дерева, і як ефективно об'єднувати ці дерева? Відповідь на це питання дає спеціалізована структура даних.

1.2. Система неперетинних множин

Уявіть, що кожне дерево в нашому лісі — це множина вершин. На початку в нас є $|V|$ множин, кожна з яких містить одну вершину. Коли ми додаємо

ребро, ми фактично об'єднуємо дві множини в одну. Перевірка, чи утворить ребро (u, v) цикл, зводиться до перевірки, чи елементи u та v вже належать до однієї множини.

Структура даних DSU ідеально підходить для цього. Вона підтримує колекцію неперетинних множин і надає дві основні операції:

- `Find(item)`: Визначити, до якої множини належить елемент `item`. Ця операція повертає унікальний "представник" (або корінь) множини.
- `Union(item1, item2)`: Об'єднати дві множини, що містять `item1` та `item2`.

На додаток до них, є ініціалізуюча операція:

- `MakeSet(item)`: Створити нову множину, що містить лише один елемент `item`.

Давайте розглянемо, як це реалізувати. Найпоширеніший спосіб — представлення кожної множини у вигляді дерева, де кожен вузол зберігає посилання лише на свого "батька". Корінь дерева є представником множини і вказує сам на себе.

Псевдокод базових операцій DSU

Для реалізації нам знадобиться масив `parent`, де `parent[i]` зберігає батька елемента `i`.

```
// Створює нову множину для елемента v
function MakeSet(v, parent):
  parent[v] = v
// Знаходить представника множини, до якої належить v (наївна версія)
function Find(v, parent):
  if v == parent[v]:
    return v
  return Find(parent[v], parent) // Рекурсивно йдемо до кореня
// Об'єднує множини, що містять u та v (наївна версія)
function Union(u, v, parent):
  root_u = Find(u, parent)
  root_v = Find(v, parent)
  if root_u != root_v:
    parent[root_v] = root_u //Просто приєднуємо одне дерево до іншого
```

Наївна реалізація `Find` та `Union` може бути неефективною. Якщо ми постійно об'єднуємо множини, дерева можуть вироджуватися в довгі ланцюжки. У такому випадку операція `Find` для елемента в кінці ланцюжка займатиме час, пропорційний кількості елементів у множині, тобто $O(N)$. Це занадто повільно. На щастя, існують дві прості, але потужні оптимізації.

1.3. Оптимізації DSU

1. Об'єднання за рангом/розміром

Ідея полягає в тому, щоб уникнути створення довгих ланцюжків. При об'єднанні двох дерев ми будемо приєднувати менше дерево до кореня більшого дерева. Це допомагає зберігати дерева "приземкуватими" і невисокими. "Розмір" можна відстежувати двома способами: за кількістю вузлів у дереві (union by size) або за висотою/глибиною дерева (union by rank). Обидва підходи ефективні. Розглянемо об'єднання за рангом.

Нам знадобиться додатковий масив `rank`, де `rank[i]` зберігає ранг (умовну висоту) дерева з коренем `i`.

Псевдокод `Union` з оптимізацією

```
function MakeSet(v, parent, rank):
  parent[v] = v
  rank[v] = 0 // Початковий ранг дерева з одного вузла - 0

function Union_by_Rank(u, v, parent, rank):
  root_u = Find(u, parent) // Find поки що наївний
  root_v = Find(v, parent)

  if root_u != root_v:
    // Приєднуємо дерево з меншим рангом до дерева з більшим рангом
    if rank[root_u] < rank[root_v]:
      parent[root_u] = root_v
    else if rank[root_u] > rank[root_v]:
      parent[root_v] = root_u
    else:
      // Ранги однакові. Приєднуємо v до u і збільшуємо ранг u.
      parent[root_v] = root_u
      rank[root_u] = rank[root_u] + 1
```

4. Стиснення шляху (Path Compression)

Ця оптимізація прискорює операцію `Find`. Коли ми шукаємо представника для елемента `v`, ми проходимо шлях від `v` до кореня. Ідея стиснення шляху полягає в тому, щоб після знаходження кореня зробити всі вузли на цьому

шляху прямими нащадками кореня. Це “сплющує” дерево, роблячи майбутні виклики `Find` для цих елементів дуже швидкими.

Псевдокод `Find` зі стисненням шляху

```
function Find_with_Path_Compression(v, parent):
  if v == parent[v]:
    return v

  // Рекурсивно знаходимо корінь і ОДНОЧАСНО
  // переприв'язуємо батька v до кореня
  root = Find_with_Path_Compression(parent[v], parent)
  parent[v] = root
  return root
```

Коли обидві оптимізації (`Union by Rank/Size` та `Path Compression`) використовуються разом, амортизована часова складність однієї операції DSU стає майже константною. Вона дорівнює $O(\alpha(N))$, де $\alpha(N)$ — обернена функція Аккермана. Ця функція росте настільки повільно, що для будь-яких практично можливих значень N (наприклад, більших за кількість атомів у видимому Всесвіті) її значення не перевищить 5. Тому на практиці складність DSU можна вважати константною.

1.4. Фінальний алгоритм Крускала

Тепер, озброївшись ефективною структурою DSU, ми можемо сформулювати повний алгоритм.

Псевдокод алгоритму Крускала

```
function KruskalMST(graph G = (V, E)):
  // 1. Ініціалізуємо МКД як порожню множину
  MST = {}

  // Ініціалізуємо DSU: кожна вершина - окрема множина
  // parent і rank - масиви для DSU
  for each vertex v in V:
    MakeSet(v, parent, rank) // Використовує оптимізовану версію

  // 2. Сортуємо всі ребра графа E за вагою в неспадаючому порядку
  sorted_edges = sort_edges_by_weight(E)

  // 3. Ітеруємо по відсортованих ребрах
  for each edge (u, v) with weight w in sorted_edges:
    // 4. Перевіряємо, чи не утворить ребро цикл
    // Використовуємо DSU з усіма оптимізаціями
    if Find(u, parent) != Find(v, parent):
```

```
        // Якщо вершини в різних множинах, додаємо ребро
        add_edge (u, v) to MST
        // І об'єднуємо їхні множини
        Union(u, v, parent, rank)

    return MST
```

Аналіз складності:

- Сортування ребер: Це домінуюча частина алгоритму. Якщо у нас $|E|$ ребер, сортування займає $O(|E|\log|E|)$.
- Ініціалізація DSU: Цикл `for` для `MakeSet` виконується $|V|$ разів, загальний час $O(|V|)$.
- Цикл по ребрах: Ми виконуємо $2 \cdot |E|$ операцій `Find` та до $|V| - 1$ операцій `Union`. З оптимізаціями DSU, загальний час для цього циклу становить приблизно $O(|E|\alpha(|V|))$.

Загальна складність алгоритму визначається найповільнішим кроком, яким є сортування: $O(|E|\log|E|)$. Оскільки в зв'язному графі $|E|$ може бути до $O(|V|^2)$, то $\log|E|$ є $O(\log(|V|^2)) = O(2\log|V|) = O(\log|V|)$. Тому складність часто записують як $O(|E|\log|V|)$, щоб було легше порівнювати з іншими графовими алгоритмами.

2. Область застосування

Оскільки алгоритм Крускала, як і алгоритм Прима, знаходить МКД, сфери їх застосування ідентичні. Алгоритм є незамінним у:

- Проектуванні будь-яких мереж: комп'ютерних, транспортних, енергетичних, де головна мета — з'єднати всі вузли з мінімальними витратами.
- Кластерному аналізі: для групування даних. Уявивши дані як вершини, а відстань між ними як вагу ребер, МКД може допомогти виявити природні скупчення.
- Біоінформатиці: для аналізу генних мереж або філогенетичних дерев.
- Наближених обчисленнях: як частина складніших алгоритмів, наприклад, для розв'язання задачі комівояжера.

Вибір між алгоритмами Прима та Крускала часто залежить від характеристик графа та особливостей реалізації, про що ми поговоримо нижче.

3. Приклад застосування

Давайте знову візьмемо той самий граф з попереднього розділу, щоб порівняти роботу двох алгоритмів.

Вхідний граф:

- Вершини: {A, B, C, D, E, F}
- Ребра та ваги: (A,B,2), (A,C,3), (A,D,3), (B,C,4), (B,E,3), (C,D,5), (C,F,6), (D,F,7), (E,F,8).

Крок 1: Сортування ребер Спочатку відсортуємо всі ребра за вагою:

2. (A, B) - 2
3. (A, C) - 3
4. (A, D) - 3
5. (B, E) - 3
6. (B, C) - 4
7. (C, D) - 5
8. (C, F) - 6
9. (D, F) - 7
10. (E, F) - 8

Крок 2: Ініціалізація DSU Створюємо 6 множин, по одній для кожної вершини: {A}, {B}, {C}, {D}, {E}, {F}. `MST` = {}.

Крок 3: Ітерація по ребрах

- Ребро (A, B), вага 2:
    - `Find(A)` повертає представника {A}. `Find(B)` повертає представника {B}.
    - Вони різні.
    - Дія: Додаємо (A, B) до `MST`. Об'єднуємо множини: `Union(A, B)`.
    - Множини: {A, B}, {C}, {D}, {E}, {F}. `MST` = {(A, B)}.

– Ребро (A, C), вага 3:
    - `Find(A)` -> {A, B}. `Find(C)` -> {C}.
    - Різні.
    - Дія: Додаємо (A, C) до `MST`. `Union(A, C)`.
    - Множини: {A, B, C}, {D}, {E}, {F}. `MST` = {(A, B), (A, C)}.
– Ребро (A, D), вага 3: (Примітка: порядок ребер з однаковою вагою не важливий)
    - `Find(A)` -> {A, B, C}. `Find(D)` -> {D}.
    - Різні.
    - Дія: Додаємо (A, D) до `MST`. `Union(A, D)`.
    - Множини: {A, B, C, D}, {E}, {F}. `MST` = {(A, B), (A, C), (A, D)}.
– Ребро (B, E), вага 3:
    - `Find(B)` -> {A, B, C, D}. `Find(E)` -> {E}.
    - Різні.
    - Дія: Додаємо (B, E) до `MST`. `Union(B, E)`.
    - Множини: {A, B, C, D, E}, {F}. `MST` = {(A, B), (A, C), (A, D), (B, E)}.
– Ребро (B, C), вага 4:
    - `Find(B)` -> {A, B, C, D, E}. `Find(C)` -> {A, B, C, D, E}.
    - Однакові! Додавання цього ребра утворить цикл (A-B-C-A).
    - Дія: Відкидаємо ребро.
– Ребро (C, D), вага 5:
    - `Find(C)` -> {A, B, C, D, E}. `Find(D)` -> {A, B, C, D, E}.
    - Однакові!
    - Дія: Відкидаємо ребро.
– Ребро (C, F), вага 6:

- `Find(C)` -> {A, B, C, D, E}. `Find(F)` -> {F}.
- Різні.
- Дія: Додаємо (C, F) до `MST`. `Union(C, F)`.
- Множини: {A, B, C, D, E, F}. `MST` = {(A, B), (A, C), (A, D), (B, E), (C, F)}.

Крок 4: Завершення Ми додали $5 = |V| - 1$ ребер. Всі вершини тепер в одній множині. Алгоритм завершено.

Результат: Мінімальне кістякове дерево складається з ребер: (A, B), (A, C), (A, D), (B, E), (C, F). Загальна вага = $2 + 3 + 3 + 3 + 6 = 17$. Як бачимо, результат ідентичний тому, що ми отримали за допомогою алгоритму Прима.

Переваги та недоліки

Переваги

– Концептуальна простота: Основна ідея “бери найдешевше ребро, що не створює циклу” дуже проста для розуміння.

– Ефективність на розріджених графах: Складність $O(E\log E)$ робить його дуже швидким для розріджених графів, де $|E|$ близьке до $|V|$. На таких графах він часто перевершує $O(V^2)$ реалізацію Прима.

– Легкість реалізації (за наявності бібліотек): Якщо у вас є готова функція сортування та структура DSU, реалізація стає тривіальною.

– Природна робота з незв’язними графами: На відміну від Прима, який потрібно запускати для кожної компоненти зв’язності, Крускал автоматично знаходить мінімальний кістяковий ліс для незв’язного графа.

Недоліки

– Високі витрати на сортування: Сортування всіх ребер може бути повільним і вимагати багато пам’яті, особливо для щільних графів. Для щільного графа $|E| \approx O(V^2)$, і складність $O(E\log E)$ стає гіршою за $O(V^2)$ у реалізації Прима.

– Необхідність у DSU: Потребує реалізації (або використання готової) структури даних DSU, яка, хоч і ефективна, є складнішою за прості масиви, що використовуються в наївній версії Прима.

– Не “онлайн” алгоритм: Потребує всіх ребер графа наперед для сортування. Алгоритм Прима, навпаки, може працювати в більш “онлайн” режимі, знаючи лише вершини, суміжні з уже побудованим деревом.

Контрольні запитання

1. У чому полягає ключова відмінність “жадібної” стратегії Крускала від стратегії Прима?
2. Яку роль виконує структура даних DSU в алгоритмі Крускала? Які дві основні операції вона надає?
3. Як за допомогою DSU визначається, чи утворить нове ребро цикл у поточній побудові?
4. Назвіть та поясніть дві оптимізації, що роблять DSU надзвичайно ефективною.
5. Чому сортування ребер є “вузьким місцем” в асимптотичній складності алгоритму Крускала?
6. В якому випадку алгоритм Крускала буде працювати значно швидше за наївну реалізацію алгоритму Прима ($O(V^2)$)?
7. Що таке “мінімальний кістяковий ліс” і чому алгоритм Крускала знаходить його автоматично?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Крускала з використанням системи неперетинних множин для пошуку МКД.

2. Мета

– Закріпити теоретичні знання про алгоритм Крускала.

– Отримати практичні навички реалізації та використання структури даних “система неперетинних множин” (DSU) з оптимізаціями.

– Порівняти підхід Крускала з підходом Прима до вирішення задачі пошуку МКД.

3. Завдання

a) Спроектувати та реалізувати клас або модуль для структури даних DSU, що підтримує операції `MakeSet`, `Find` (зі стисненням шляху) та `Union` (з об'єднанням за рангом або розміром).

b) Реалізувати алгоритм Крускала, який використовує вашу структуру DSU.

c) Представлення графа може бути простим списком ребер, оскільки алгоритму не потрібні списки суміжності.

d) Забезпечити завантаження графа з текстового файлу (формат: `u v w`).

e) Програма повинна виводити на екран ребра, що увійшли до МКД, та його сумарну вагу.

4. Опис

Лабораторна робота фокусується на двох основних аспектах: реалізації ефективної структури даних DSU та її застосуванні в алгоритмі Крускала. Студент повинен продемонструвати розуміння того, як DSU допомагає вирішити проблему виявлення циклів, і як сортування ребер визначає жадібну стратегію алгоритму.

5. Порядок виконання

a) Проектування та реалізація DSU. Це перший і найважливіший крок. Створіть окремий клас/модуль для DSU. Переконайтеся, що реалізовані обидві оптимізації: стиснення шляху та об'єднання за рангом/розміром. Ретельно протестуйте його на простих прикладах.

b) Реалізація сортування. Використовуйте вбудовану в мову програмування функцію сортування для списку/масиву ребер. Ребра потрібно зберігати у структурах або класах, щоб можна було сортувати їх за полем ваги.

c) Реалізація алгоритму Крускала. Напишіть головну функцію алгоритму, яка ініціалізує DSU, сортує ребра та ітерує по них, викликаючи операції `Find` та `Union`.

d) Інтеграція. З'єднайте всі частини: читання графа з файлу, сортування, застосування алгоритму та виведення результату.

e) Тестування. Перевірте роботу програми на різних графах: розріджених, щільних, незв'язних. Порівняйте результати з тими, що дає алгоритм Прима (вони мають бути однаковими за вагою).

f) Написання звіту. Оформіть звіт, приділивши особливу увагу опису реалізації DSU та її оптимізацій.

6. Результат

Результатом роботи є програмний продукт, що знаходить МКД (або мінімальний кістяковий ліс) за алгоритмом Крускала, та звіт, що пояснює реалізацію та її компоненти.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
    - Реалізація структури даних DSU з оптимізаціями.
    - Основна логіка алгоритму Крускала.
    - Структури для представлення ребер та графа.
2. Продемонструвати роботу додатку:
    - На тестових даних, що включають граф з кількома компонентами зв'язності.
    - На великому розрідженому графі, щоб продемонструвати ефективність.
    - Порівняти результат (сумарну вагу) з результатом роботи програми з попередньої лабораторної роботи (алгоритм Прима).
3. Бути готовим відповісти на питання:
    - Питання з розділу "Контрольні запитання".

- Пояснити на прикладі, як працює стиснення шляху у вашій реалізації DSU.
- Чому для алгоритму Крускала достатньо представити граф як список ребер, на відміну від Прима, де зручніші списки суміжності?
- Що станеться, якщо не сортувати ребра перед виконанням алгоритму?
- Яка асимптотична складність операцій DSU у вашому коді?

Тести

1. Який перший крок виконується в алгоритмі Крускала?
    1. Вибір довільної стартової вершини.
    2. Побудова черги з пріоритетом для вершин.
    3. Сортування всіх ребер графа за вагою.
    4. Ініціалізація DSU для всіх ребер.
2. Яка структура даних є центральною для ефективної реалізації алгоритму Крускала?
    1. Черга з пріоритетом (Priority Queue).
    2. Хеш-таблиця (Hash Table).
    3. Двійкове дерево пошуку (Binary Search Tree).
    4. Система неперетинних множин (Disjoint Set Union).
3. Як алгоритм Крускала визначає, що додавання ребра (u, v) утворить цикл?
    1. Якщо `dist[u]` + `dist[v]` менше за вагу ребра.
    2. Якщо вершини `u` та `v` вже з'єднані іншим ребром.
    3. Якщо операція `Find(u)` повертає того ж представника, що й `Find(v)`.
    4. За допомогою пошуку в глибину після додавання ребра.

4. Для якого типу графів алгоритм Крускала ($O(E\log E)$), як правило, є більш ефективним?
    1. Для щільних графів.
    2. Для повних графів.
    3. Для розріджених графів.
    4. Для графів з від’ємними вагами.
5. Що знайде алгоритм Крускала, якщо його запустити на незв’язному графі?
    1. МКД для найбільшої компоненти зв’язності.
    2. Помилку, оскільки граф незв’язний.
    3. Мінімальний кістяковий ліс.
    4. Порожню множину ребер.

Відповіді на тести 1. - 3), 2. - 4), 3. - 3), 4. - 3), 5. - 3),

Тести підвищеної складності

1. Який компонент домінує в асимптотичній складності алгоритму Крускала?
    1. Ініціалізація структури DSU.
    2. Сортування ребер.
    3. Послідовність операцій `Find/Union`.
    4. Побудова результуючого дерева.
2. Оптимізація “стиснення шляху” в DSU покращує асимптотичну складність якої операції?
    1. `Union`
    2. `MakeSet`
    3. `Find`
    4. Усіх операцій однаково.

3. Якщо в графі є декілька ребер з однаковою мінімальною вагою, який вибір зробить алгоритм Крускала?
    1. Вибере перше з них, яке зустрінеться у відсортованому масиві.
    2. Вибере те з них, яке з'єднує компоненти меншого розміру.
    3. Видасть помилку через неоднозначність.
    4. Вибере всі такі ребра, якщо вони не утворюють циклів.
4. "Жадібний" вибір в алгоритмі Прима — це ___, а в алгоритмі Крускала — це ___.
    1. …найлегше ребро з усього графа / …найлегше ребро до сусідньої вершини.
    2. …найлегше ребро, що виходить з побудованого дерева / …найлегше ребро в усьому графі, що не створює цикл.
    3. …вершина, найближча до дерева / …ребро, найближче до центру графа.
    4. Обидва алгоритми роблять однаковий жадібний вибір.
5. Якщо реалізувати DSU без оптимізацій (дерева можуть вироджуватися в списки), яка буде найгірша складність виконання циклу з операціями `Find/Union` в алгоритмі Крускала?
    1. $O(E \log V)$
    2. $O(E \cdot V)$
    3. $O(E + V)$
    4. $O(E\alpha(V))$
6. Для знаходження максимального кістякового дерева за допомогою алгоритму Крускала потрібно:
    1. Сортувати ребра за спаданням ваги.
    2. Інвертувати ваги ребер (помножити на -1) і сортувати за зростанням.

3. Використовувати максимальну чергу з пріоритетом замість сортування.

4. Правильні відповіді 1 і 2.

7. У чому полягає оптимізація “об’єднання за рангом”?

    1. Завжди приєднувати множину з меншою кількістю елементів до більшої.

    2. Завжди приєднувати дерево з меншою висотою (рангом) до кореня дерева з більшою висотою.

    3. Завжди робити представником елемент з найбільшим індексом.

    4. Об’єднувати лише ті елементи, ранг яких відрізняється не більше ніж на 1.

8. Чи можна реалізувати алгоритм Крускала, використовуючи замість DSU пошук в глибину (DFS) або в ширину (BFS) для перевірки наявності циклу на кожному кроці?

    1. Ні, це неможливо.

    2. Так, але це буде значно повільніше.

    3. Так, і це буде так само ефективно.

    4. Це можливо тільки з BFS, але не з DFS.

9. Складність $O(E\alpha(V))$ для операцій DSU є:

    1. Складністю у найгіршому випадку.

    2. Амортизованою складністю.

    3. Складністю у середньому випадку.

    4. Складністю у найкращому випадку.

10. Алгоритм Крускала є прикладом загального методу побудови МКД, заснованого на:

    1. Цикловому правилі (видалення найважчого ребра з циклу).

2. Правилі розрізу (додавання найлегшого ребра, що перетинає розріз).
3. Обидва правила лежать в основі алгоритму.
4. Жодне з цих правил не застосовується.

Відповіді на тести підвищеної складності

1. 2). Сортування ($O(E\log E)$) має вищу асимптотичну складність, ніж майже лінійний час ($O(E\alpha(V))$) для операцій DSU.

2. 3). Стиснення шляху безпосередньо оптимізує пошук кореня, роблячи дерева DSU пласкими, що прискорює наступні виклики `Find`.

3. 1). Алгоритм є детермінованим. Він обробить ребра в тому порядку, в якому їх розташує функція сортування.

4. 2). Прим робить локальний вибір (найлегше ребро з "відомого" світу), Крускал — глобальний (найлегше ребро з усіх можливих, що задовольняє умові).

5. 2). Без оптимізацій кожна операція `Find` у найгіршому випадку може обійти $O(V)$ вершин. Виконуючи це для $O(E)$ ребер, отримуємо складність $O(E \cdot V)$.

6. 4). Сортування за спаданням змусить алгоритм вибирати найважчі ребра першими. Множення на -1 перетворює задачу пошуку максимуму на еквівалентну задачу пошуку мінімуму.

7. 2). Це визначення об'єднання за рангом, яке мінімізує висоту результуючого дерева.

8. 2). Так, на кожному кроці можна додати ребро до поточного лісу і запустити DFS/BFS, щоб перевірити, чи знаходяться кінці ребра в одній компоненті зв'язності. Але складність такої перевірки ($O(V + E)$) зробить весь алгоритм дуже повільним ($O(E \cdot (V + E))$).

9. 2). Це амортизована складність, тобто середня вартість операції в довгій послідовності операцій, навіть якщо деякі окремі операції можуть бути дорожчими.

10. 2). Алгоритм Крускала є канонічним прикладом застосування правила розрізу. На кожному кроці він розглядає розріз між множиною вершин одного дерева та рештою графа і додає найлегше ребро, що його перетинає (яким виявляється поточне ребро з відсортованого списку).

Завдання для самостійної роботи

1. Реалізуйте структуру DSU без оптимізацій. Порівняйте час її роботи на великих даних з версією, що має обидві оптимізації.
2. Напишіть програму, яка візуалізує роботу алгоритму Крускала: показує відсортований список ребер і покроково підсвічує ребра, що додаються до МКД, та ті, що відкидаються.
3. Модифікуйте алгоритм Крускала так, щоб він знаходив не МКД, а “пляшкове горло” графа — дерево, в якому максимальна вага ребра є мінімально можливою.
4. Доведіть, що алгоритм Крускала завжди будує дерево (тобто зв’язний ациклічний граф).
5. Дослідіть, як зміниться МКД, якщо з графа видалити ребро, яке належало початковому МКД. Чи потрібно запускати алгоритм заново?
6. Реалізуйте задачу “реверсивного” алгоритму для пошуку МКД: почніть з усіма ребрами і видаляйте найважчі ребра, які не порушують зв’язності графа. Порівняйте його з алгоритмом Крускала.
7. Що станеться, якщо у вашому DSU ви будете використовувати тільки стиснення шляху, але не об’єднання за рангом? Дослідіть теоретичну складність та перевірте на практиці.
8. Задача: є N міст і вартість побудови дороги між будь-якими двома. Деякі дороги вже побудовані. Потрібно з’єднати всі міста з мінімальними додатковими витратами. Сформулюйте цю задачу в термінах МКД і розв’яжіть її.
9. Напишіть функцію, яка приймає граф і набір ребер, і перевіряє, чи є цей набір ребер мінімальним кістяковим лісом для цього графа.

10. Адаптуйте алгоритм Крускала для роботи з мультиграфом (де між двома вершинами може бути кілька ребер).

Теми лабораторних робіт

1. Реалізація алгоритму Крускала та порівняльний аналіз продуктивності з алгоритмом Прима на щільних та розріджених графах.
2. Створення інтерактивного візуалізатора для алгоритмів Прима та Крускала.
3. Використання DSU для задач, не пов’язаних з МКД (наприклад, визначення кількості компонент зв’язності в динамічному графі).
4. Розробка програми для розрахунку мінімальної вартості об’єднання існуючих локальних мереж в одну глобальну.
5. “Реверсивний” алгоритм Крускала: реалізація та порівняння.
6. Застосування алгоритму Крускала для автоматичної генерації лабіринтів.
7. Побудова МКД для графа соціальної мережі, де вага ребра — це “відстань” між користувачами.
8. Дослідження впливу різних евристик (Union by Size vs Union by Rank) на продуктивність DSU.
9. Реалізація алгоритму для пошуку k-го найкращого кістякового дерева.
10. Алгоритм Крускала для графів, що зберігаються у зовнішній пам’яті (коли граф не вміщується в RAM).
11. Паралельна реалізація алгоритму Крускала.
12. Розробка модуля для системи автоматизованого проектування (САПР) для оптимізації розводки провідників на платі з використанням МКД.

Максимальний потік / Мінімальний розріз.

Якщо в попередніх розділах ми вчилися з'єднувати всі вершини мережі з мінімальними витратами (МКД), то тепер ми зосередимося на тому, як максимально ефективно використовувати вже існуючу мережу.

Уявіть собі мережу трубопроводів, що з'єднує джерело води зі споживачем. Кожна труба має обмежену пропускну здатність — максимальний об'єм води, який може пройти через неї за одиницю часу.

Наше завдання — визначити, яку максимальну кількість води ми можемо перекачати від джерела до споживача, не перевантажуючи жодну з труб. Це і є класична задача про максимальний потік.

Ця, на перший погляд, інженерна задача має дивовижно глибокий зв'язок з іншою концепцією — мінімальним розрізом. Розріз — це уявна "лінія", що розділяє всі вершини мережі на дві частини, де джерело s знаходиться в одній частині, а стік t — в іншій. Пропускна здатність такого розрізу — це сумарна пропускна здатність труб, що ведуть з частини з джерелом до частини зі стоком. Мінімальний розріз відповідає "найвужчому місцю" або "пляшковому горлу" всієї мережі.

Центральною перлиною цієї теми є фундаментальна теорема про максимальний потік і мінімальний розріз, яка стверджує, що ці два значення завжди рівні! Значення максимального потоку, який можна пропустити через мережу, точно дорівнює пропускній здатності її найвужчого місця.

У цьому підрозділі ми розглянемо кілька ключових алгоритмів для розв'язання цієї задачі, кожен з яких є еволюцією попереднього або пропонує кардинально новий підхід:

- Метод Форда-Фалкерсона: Загальна ідея, що лежить в основі багатьох потокових алгоритмів. Ми будемо ітеративно знаходити "шляхи для покращення" потоку від s до t і збільшувати потік вздовж них.
- Алгоритм Едмондса-Карпа: Конкретна реалізація методу Форда-Фалкерсона, яка використовує пошук в ширину (BFS) для знаходження найкоротших шляхів покращення, що гарантує завершення роботи та має чітко визначену складність.

– Алгоритм Дініца: Значно складніший та ефективніший підхід, який використовує концепцію “допоміжної шаруватої мережі” та “блокуючих потоків” для знаходження багатьох шляхів покращення за одну фазу.

– Метод “проштовхування-перепозначення”: Кардинально інший підхід, що не шукає шляхи від s до t. Він моделює потік як фізичний процес, де вершини можуть мати “надлишковий” потік, який вони “проштовхують” до своїх сусідів, доки система не досягне рівноваги, що відповідає максимальному потоку.

Максимальний потік. Метод Форда-Фалкерсона

У вступі до цієї великої теми ми познайомилися з інтуїтивним поняттям потоку в мережі. Тепер настав час перейти від інтуїції до формалізму та алгоритмів. Першим і найбільш фундаментальним підходом, який ми розглянемо, є метод Форда-Фалкерсона. Важливо розуміти, що це не стільки конкретний алгоритм, скільки загальна методологія або фреймворк для розв’язання задачі про максимальний потік. Його основна ідея полягає у фразі: “поки можемо збільшити потік, будемо його збільшувати”.

Перш ніж зануритись в сам метод, нам потрібно формалізувати ключові поняття, з якими ми будемо працювати.

1.1. Потокова мережа та потік

Потокова мережа (Flow Network) — це орієнтований граф $G = (V, E)$, в якому:

1. Виділено дві особливі вершини: джерело (source) s та стік (sink) t.
2. Кожне ребро $(u, v) \in E$ має невід’ємну пропускну здатність (capacity) $c(u, v) \geq 0$. Якщо ребра (u, v) не існує, ми вважаємо, що $c(u, v) = 0$.

Потік (Flow) у мережі G — це функція $f: V \times V \to \mathbb{R}$, що для кожної пари вершин (u, v) визначає величину потоку, який тече по ребру (u, v). Ця функція повинна задовольняти трьом умовам:

3. Обмеження пропускної здатності: Потік по будь-якому ребру не може перевищувати його пропускну здатність.

$$\forall u, v \in V: \quad f(u, v) \leq c(u, v)$$

4. Антисиметричність: Потік з u до v є протилежним до потоку з v до u. Це формальна вимога, яка спрощує математичні доведення.

$$\forall u, v \in V: \quad f(u, v) = -f(v, u)$$

На практиці це означає, що якщо ми пускаємо 5 одиниць потоку з u в v, то ми формально вважаємо, що з v в u тече -5 одиниць.

5. Збереження потоку: Для будь-якої вершини, крім джерела та стоку, сумарний потік, що входить у вершину, повинен дорівнювати сумарному потоку, що виходить з неї. Іншими словами, проміжні вершини не виробляють і не поглинають потік.

$$\forall u \in V \backslash \{s, t\}: \quad \sum_{v \in V} f\,(u, v) = 0$$

Величина потоку, що позначається як $|f|$, — це сумарний потік, що виходить з джерела s (або, що еквівалентно, входить у стік t).

$$|f| = \sum_{v \in V} f\,(s, v)$$

Задача про максимальний потік полягає у знаходженні такого потоку f, для якого величина $|f|$ є максимально можливою.

1.2. Залишкова мережа та шляхи, що доповнюють потік

Основна ідея методу Форда-Фалкерсона полягає в ітеративному покращенні потоку. Ми починаємо з нульового потоку і на кожному кроці намагаємося знайти шлях від s до t, по якому можна пустити ще трохи потоку. Цей процес вимагає введення двох критично важливих понять.

Залишкова мережа $G_f = (V, E_f)$

Для потоку f у мережі G залишкова мережа G_f показує, скільки ще потоку ми можемо "проштовхнути" по кожному ребру. Її ребра E_f визначаються залишковою пропускною здатністю (residual capacity) $c_f(u, v)$:

$$c_f(u, v) = c(u, v) - f(u, v)$$

Це значення показує, скільки "вільного місця" залишилось у ребрі (u, v). Залишкова мережа складається з усіх ребер (u, v), для яких $c_f(u, v) > 0$.

Тут є один надзвичайно важливий нюанс. Коли ми пускаємо потік $f(u, v)$ по ребру (u, v), ми не лише зменшуємо його пряму пропускну здатність, але й створюємо можливість "скасувати" цей потік. Це моделюється появою зворотного ребра (backward edge) (v, u) в залишковій мережі. Його залишкова пропускна здатність дорівнює величині потоку, який ми пустили в прямому напрямку: $c_f(v, u) = f(u, v)$.

Отже, залишкова мережа G_f для потоку f містить:

1. Прямі ребра: Усі ребра (u, v) з оригінального графа, де потік ще не досяг максимуму. Їхня залишкова пропускна здатність $c_f(u, v) = c(u, v) - f(u, v) > 0$.
2. Зворотні ребра: Для кожного ребра (u, v), по якому тече потік $f(u, v) > 0$, з'являється ребро (v, u) із залишковою пропускною здатністю $c_f(v, u) = f(u, v)$.

Навіщо потрібні зворотні ребра? Вони дають алгоритму гнучкість. Проштовхування потоку по зворотному ребру (v, u) еквівалентне зменшенню потоку по прямому ребру (u, v). Це дозволяє алгоритму виправляти свої "помилкові" жадібні рішення. Можливо, раніше ми направили потік по шляху, який виявився неоптимальним, а тепер, використовуючи зворотне ребро, ми можемо "перенаправити" частину цього потоку на кращий шлях.

Шлях, що доповнює потік

Це будь-який простий шлях від s до t у залишковій мережі G_f. Якщо такий шлях існує, це означає, що ми можемо збільшити наш загальний потік.

Залишкова пропускна здатність шляху $c_f(p)$ — це мінімальна залишкова пропускна здатність серед усіх ребер, що утворюють шлях p.

$$c_f(p) = \min\{c_f(u, v) \mid (u, v) \text{ є ребром у шляху } p\}$$

Це значення показує, скільки додаткового потоку ми можемо "протиснути" через цей шлях — він обмежений його "найвужчим місцем".

1.3. Метод Форда-Фалкерсона

Тепер ми готові сформулювати сам метод.

Псевдокод методу Форда-Фалкерсона

```
function FordFulkerson(G, s, t):
  // 1. Ініціалізувати потік f нулем для всіх ребер (u,v)
  for each edge (u, v) in G.E:
    f(u, v) = 0
    f(v, u) = 0

  // 2. Поки існує шлях p від s до t у залишковій мережі G_f
  while there exists a path p from s to t in the residual network G_f
:
    // a. Знайти залишкову пропускну здатність цього шляху
```

```
    cfp = min_residual_capacity(p) // cfp - capacity of flow path

    // b. Збільшити потік f
    for each edge (u, v) in path p:
      // Якщо (u,v) - пряме ребро в G
      if (u,v) is a forward edge:
        f(u, v) = f(u, v) + cfp
      // Якщо (u,v) - зворотне ребро до (v,u) в G
      else: // (u,v) is a backward edge, so we decrease flow on (v,u)
        f(v, u) = f(v, u) - cfp

    // Примітка: оновлення f автоматично оновлює G_f для наступної ітерації

  // 3. Коли шляхів більше немає, потік f є максимальним
  return f
```

Метод є загальним, оскільки він не специфікує, як саме знаходити шлях, що доповнює потік. Для цього можна використовувати будь-який алгоритм пошуку шляху, наприклад, пошук в глибину (DFS) або пошук в ширину (BFS).

Псевдокод пошуку шляху за допомогою DFS

```
function FindAugmentingPath_DFS(G_f, start_node, end_node):
  // visited - множина відвіданих вершин для уникнення циклів
  visited = {}
  // path_stack - стек для відновлення шляху
  path_stack = new Stack()
  path_stack.push(start_node)

  // parent_map - словник для відновлення шляху
  parent_map = {}

  while not path_stack.is_empty():
    u = path_stack.pop()

    if u == end_node:
      // Ми знайшли шлях, відновлюємо його
      return reconstruct_path(parent_map, start_node, end_node)

    if u not in visited:
      visited.add(u)
      // Розглядаємо всіх сусідів v вершини u в залишковій мережі
      for each neighbor v of u in G_f:
        if v not in visited and G_f.capacity(u, v) > 0:
          parent_map[v] = u
```

```
            path_stack.push(v)

    // Якщо цикл завершився, а шлях не знайдено
    return NULL // Шляху не існує
```

1.4. Аналіз складності та коректності

Коректність: Коли алгоритм завершується, це означає, що в залишковій мережі більше немає шляху від s до t. Це, в свою чергу, є критерієм досягнення максимального потоку згідно з теоремою про максимальний потік і мінімальний розріз.

Складність: Тут ситуація складніша.

- Якщо всі пропускні здатності — цілі числа, то на кожному кроці величина потоку збільшується щонайменше на 1. Оскільки максимальний потік $|f_{max}|$ обмежений, алгоритм гарантовано завершиться. Час роботи однієї ітерації (пошук шляху) становить $O(E)$ (для DFS/BFS). Таким чином, загальна складність — $O(|f_{max}| \cdot E)$.
- Проблема: Ця складність може бути дуже поганою. Якщо $|f_{max}|$ — дуже велике число, алгоритм працюватиме довго. Існують спеціально сконструйовані графи, де невдалий вибір шляхів (наприклад, “довгих” і “тонких” замість “коротких” і “широких”) призводить до величезної кількості ітерацій.
- Якщо пропускні здатності — дійсні (ірраціональні) числа, метод Форда-Фалкерсона може працювати нескінченно довго, так і не досягнувши максимального значення потоку.

Саме через цю залежність від величини потоку та невдалого вибору шляхів, Форд-Фалкерсон розглядається як *метод*, а не як ефективний *алгоритм*. Специфічні реалізації, як-от алгоритм Едмондса-Карпа (де шлях шукається за допомогою BFS), вирішують цю проблему і мають поліноміальну складність.

2. Область застосування

Задачі, що зводяться до пошуку максимального потоку, зустрічаються у багатьох, іноді несподіваних, сферах:

- Логістика та транспорт: Розрахунок максимального обсягу вантажу, який можна перевезти з пункту А в пункт Б через транспортну мережу з обмеженою пропускною здатністю доріг.

– Телекомунікації та комп’ютерні мережі: Визначення максимальної швидкості передачі даних між двома вузлами в мережі.

– Планування виробництва: Розрахунок максимальної кількості продукції, яку можна виробити, враховуючи обмежені потужності різних етапів виробничого ланцюга.

– Задача про призначення (Assignment Problem): Наприклад, пошук максимальної кількості пар у дводольному графі (bipartite graph). Ця задача зводиться до пошуку максимального потоку в спеціально побудованій мережі, де ребра мають одиничну пропускну здатність.

– Комп’ютерний зір: Сегментація зображень, де потрібно розділити об’єкт і фон. Задача може бути сформульована як задача про мінімальний розріз, яка, як ми знаємо, еквівалентна задачі про максимальний потік.

3. Приклад застосування

Розглянемо покроково роботу методу на простому прикладі. Нехай є мережа:

– Вершини: S, A, B, T

– Ребра та пропускні здатності: c(S,A)=10, c(S,B)=10, c(A,B)=2, c(A,T)=10, c(B,T)=10.

Крок 0: Ініціалізація. Весь потік f дорівнює 0. $|f| = 0$.

Ітерація 1:

3. Пошук шляху: Шукаємо шлях в залишковій мережі (яка наразі ідентична вихідній). Нехай DFS знаходить шлях $p_1 = S \to A \to T$.
4. Пропускна здатність шляху: $c_f(p_1) = \min(c(S, A), c(A, T)) = \min(10,10) = 10$.
5. Збільшення потоку:
   - $f(S, A) = 0 + 10 = 10$.
   - $f(A, T) = 0 + 10 = 10$.
6. Результат: Загальний потік $|f| = 10$.

Ітерація 2:

7. Будуємо залишкову мережу G_{f1}:
    - Ребро (S,A) повністю насичене, його пряма залишкова пропускна здатність 0. Але з'являється зворотне ребро (A,S) з $c_f(A,S) = f(S,A) = 10$.
    - Ребро (A,T) також насичене. З'являється зворотне ребро (T,A) з $c_f(T,A) = f(A,T) = 10$.
    - Інші ребра (S,B), (A,B), (B,T) залишаються з початковою пропускною здатністю.
8. Пошук шляху: Шукаємо шлях в G_{f1}. Нехай DFS знаходить шлях $p_2 = S \to B \to T$.
9. Пропускна здатність шляху: $c_f(p_2) = \min(c_f(S,B), c_f(B,T)) = \min(10,10) = 10$.
10. Збільшення потоку:
    - $f(S,B) = 0 + 10 = 10$.
    - $f(B,T) = 0 + 10 = 10$.
11. Результат: Загальний потік $|f| = f(S,A) + f(S,B) = 10 + 10 = 20$.

Ітерація 3:

12. Будуємо залишкову мережу G_{f2}: Тепер ребра (S,A), (A,T), (S,B), (B,T) насичені. З'явилися відповідні зворотні ребра. Ребро (A,B) все ще має залишкову пропускну здатність 2.
13. Пошук шляху: Шукаємо шлях. DFS не може знайти шлях S-A-T або S-B-T. Але він може знайти шлях, що використовує зворотне ребро! Наприклад, $p_3 = S \to A \to B \to T$ не працює, бо (A,T) насичене. А як щодо $p_3 = S \to B \to A \to T$? Це неможливо, бо в залишковій мережі немає ребра (B,A). Давайте перевіримо ще раз. Після ітерації 2, стан потоків: $f(S,A) = 10, f(A,T) = 10, f(S,B) = 10, f(B,T) = 10$. Чи є шлях в G_{f2}?
    - З S можна потрапити в A? Ні, $c_f(S,A) = 10 - 10 = 0$.
    - З S можна потрапити в B? Ні, $c_f(S,B) = 10 - 10 = 0$. Шляху від S до T в залишковій мережі більше не існує.

Завершення: Алгоритм зупиняється. Максимальний потік дорівнює 20.

А що, якби ми вибрали інший шлях на першому кроці? Нехай на ітерації 1 ми б вибрали шлях $p'_1 = S \to A \to B \to T$.

14. $c_f(p'_1) = \min(c(S,A), c(A,B), c(B,T)) = \min(10,2,10) = 2$.

15. Збільшуємо потік на 2 по цьому шляху. $|f| = 2$. Потім нам знадобилося б ще декілька ітерацій, щоб досягти 20, і на одній з них ми б точно використали зворотне ребро, щоб "перекинути" частину потоку з ребра (A,B). Саме цей механізм гарантує коректність.

Переваги та недоліки

Переваги

- Фундаментальність та простота: Ідея ітеративного покращення потоку є дуже інтуїтивною і лежить в основі багатьох складніших алгоритмів.
- Гнучкість: Метод не прив'язаний до конкретного способу пошуку шляху, що дозволяє адаптувати його (наприклад, використовуючи BFS, щоб отримати алгоритм Едмондса-Карпа).
- Гарантоване завершення для цілих чисел: Для широкого класу практичних задач, де пропускні здатності є цілими, метод гарантовано знайде оптимальне рішення.

Недоліки

- Погана продуктивність у найгіршому випадку: Складність $O(|f_{max}| \cdot E)$ робить метод непридатним для мереж з великими значеннями пропускних здатностей.
- Проблема з дійсними числами: Метод може не завершити роботу, якщо пропускні здатності є ірраціональними, що є серйозним теоретичним недоліком.
- Залежність від вибору шляху: Продуктивність сильно залежить від евристики пошуку доповнюючого шляху. Поганий вибір може призвести до значного збільшення кількості ітерацій.

Контрольні запитання

1. Що таке "залишкова мережа" і яку інформацію вона містить?

2. Поясніть, навіщо в залишковій мережі потрібні “зворотні ребра”. Яку роль вони відіграють?
3. Що таке “шлях, що доповнює потік”, і як його наявність пов’язана з можливістю збільшити потік?
4. Чому метод Форда-Фалкерсона вважається “методом”, а не “алгоритмом”?
5. Поясніть, чому складність методу Форда-Фалкерсона $O(|f_{max}| \cdot E)$ може бути дуже поганою. Наведіть ідею прикладу, де це проявляється.
6. Що станеться, якщо запустити метод на мережі, де пропускні здатності є дійсними числами?
7. Як пов’язане завершення роботи методу Форда-Фалкерсона з теоремою про максимальний потік і мінімальний розріз?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація методу Форда-Фалкерсона для пошуку максимального потоку в мережі.

2. Мета

- Закріпити теоретичні знання про задачу максимального потоку.
- Набути практичних навичок у реалізації методу Форда-Фалкерсона.
- Навчитися працювати з концепцією залишкової мережі.
- Реалізувати алгоритм пошуку шляху (DFS або BFS) як компонент основного методу.

3. Завдання

a) Спроектувати та реалізувати структуру даних для представлення потокової мережі (орієнтований граф з пропускними здатностями ребер). Рекомендується використовувати матрицю суміжності або списки суміжності.
b) Реалізувати метод Форда-Фалкерсона.

c) В рамках методу реалізувати функцію пошуку доповнюючого шляху в залишковій мережі. Для цього можна використати пошук в глибину (DFS).
d) Програма повинна на вхід отримувати опис мережі з файлу (наприклад, кількість вершин, номери джерела і стоку, а потім список ребер з пропускними здатностями).
e) На виході програма повинна видавати величину максимального потоку. Додатково можна вивести фінальний потік для кожного ребра.

4. Опис

Лабораторна робота полягає у створенні програми, що обчислює максимальний потік від заданого джерела до стоку в потоковій мережі. Ключовими елементами реалізації є коректне представлення залишкової мережі та ітераційний процес пошуку доповнюючих шляхів і збільшення потоку.

5. Порядок виконання

a) Вибір представлення графа. Оберіть, як зберігати граф: матрицею пропускних здатностей `capacity[V][V]` чи списками суміжності. Для представлення потоку знадобиться аналогічна структура `flow[V][V]`.
b) Побудова залишкової мережі. Замість того, щоб явно будувати залишкову мережу на кожному кроці, її можна представляти "віртуально". Функція пошуку шляху може обчислювати залишкову пропускну здатність "на льоту": $c_f(u, v) = c(u, v) - f(u, v)$.
c) Реалізація пошуку шляху. Напишіть функцію (наприклад, на основі DFS), яка приймає на вхід поточний стан потоку `f`, граф `G`, джерело `s` і стік `t`, і повертає шлях, що доповнює потік, або `NULL`, якщо такого немає. Функція повинна вміти рухатись як по прямих, так і по зворотних ребрах.
d) Реалізація основного циклу. Напишіть головний цикл методу Форда-Фалкерсона, який викликає функцію пошуку шляху, обчислює залишкову пропускну здатність шляху та оновлює матрицю потоку `flow`.
e) Тестування. Створіть кілька тестових мереж, включаючи приклад з теоретичної частини, та перевірте коректність роботи програми.

6. Результат

Результатом роботи є програма, що коректно обчислює максимальний потік для заданої мережі, та звіт, що описує структуру програми та прийняті рішення щодо реалізації.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
    - Структура даних для представлення мережі, потоку та пропускних здатностей.
    - Функція пошуку доповнюючого шляху (наприклад, DFS).
    - Основний цикл методу Форда-Фалкерсона.
2. Продемонструвати роботу додатку:
    - На кількох тестових прикладах, включаючи ті, що вимагають використання зворотних ребер.
    - Пояснити, як змінюється потік та залишкова мережа на кожній ітерації для невеликого прикладу.
3. Бути готовим відповісти на питання:
    - Питання з розділу “Контрольні запитання”.
    - Який алгоритм пошуку шляху ви використали і чому?
    - Як ваша програма обробляє зворотні ребра в залишковій мережі?
    - Яка складність вашої реалізації? Як вона залежить від значень пропускних здатностей?
    - Як можна було б модифікувати вашу програму для реалізації алгоритму Едмондса-Карпа?

Тести

1. Що таке “залишкова пропускна здатність” прямого ребра (u, v)?
    1. $c(u, v) + f(u, v)$
    2. $c(u, v) - f(u, v)$

3. $f(u, v)$
4. $c(u, v)$

2. Що таке “шлях, що доповнює потік” (augmenting path)?
    1. Будь-який шлях від джерела до стоку в оригінальному графі.
    2. Найкоротший шлях від джерела до стоку.
    3. Шлях від джерела до стоку в залишковій мережі.
    4. Шлях, що містить ребра з максимальним потоком.
3. Ключова перевага використання зворотних ребер у залишковій мережі полягає в тому, що вони:
    1. Дозволяють зменшити кількість ітерацій.
    2. Дозволяють алгоритму виправляти попередні “жадібні” рішення щодо направлення потоку.
    3. Спрощують пошук шляху.
    4. Гарантують завершення роботи алгоритму для дійсних чисел.
4. Часова складність методу Форда-Фалкерсона для цілочисельних пропускних здатностей дорівнює:
    1. $O(V \cdot E)$
    2. $O(E^2)$
    3. $O(|f_{max}| \cdot E)$
    4. $O(V + E)$
5. Метод Форда-Фалкерсона гарантовано знайде максимальний потік, якщо:
    1. Граф не містить циклів.
    2. Всі пропускні здатності є цілими числами.
    3. Граф є дводольним.
    4. Завжди гарантовано знаходить.

Відповіді: 1. - 2), 2. - 3), 3. - 2), 4. - 3), 5. - 2)

Тести підвищеної складності

1. В чому полягає основна відмінність між “методом Форда-Фалкерсона” та “алгоритмом Едмондса-Карпа”?
    1. Едмондс-Карп використовує матрицю суміжності, а Форд-Фалкерсон — списки.
    2. Едмондс-Карп працює для дійсних чисел, а Форд-Фалкерсон — ні.
    3. Едмондс-Карп конкретизує пошук шляху за допомогою BFS, а Форд-Фалкерсон — ні.
    4. Немає відмінностей, це різні назви одного й того ж.
2. Якщо на деякій ітерації потік був пущений по зворотному ребру (v, u), це еквівалентно:
    1. Збільшенню потоку по ребру (u, v) в оригінальній мережі.
    2. Створенню нового ребра (v, u) в оригінальній мережі.
    3. Зменшенню потоку по ребру (u, v) в оригінальній мережі.
    4. Обміну потоків місцями між (u, v) та (v, u).
3. Розглянемо мережу, де $s \to u \to t$ з $c(s, u) = 1$ і $c(u, t) = 1$, і $s \to v \to t$ з $c(s, v) = 1000$ і $c(v, t) = 1000$. Яку проблему для Форда-Фалкерсона з DFS може створити ця мережа?
    1. Ніякої, він знайде потік за 2 ітерації.
    2. DFS може постійно вибирати “поганий” шлях $s \to v \to u \to t$.
    3. DFS може постійно чергувати шляхи через u і v, збільшуючи потік лише на 1 за раз.
    4. Алгоритм може зациклитися, якщо DFS буде вибирати шлях через u, а потім через v.
4. Теорема про максимальний потік і мінімальний розріз стверджує, що потік є максимальним, якщо і тільки якщо:
    1. Всі ребра, що виходять з джерела, насичені.

2. В залишковій мережі не існує шляху від джерела до стоку.
3. Кількість ітерацій перевищила кількість вершин.
4. Всі шляхи від джерела до стоку мають однакову довжину.

5. Як можна звести задачу про пошук максимального паросполучення в дводольному графі $G = (U \cup V, E)$ до задачі про максимальний потік?
    1. Створити мережу, додавши джерело s і стік t; додати ребра з s до всіх вершин з U і з усіх вершин з V до t.
    2. Вважати одну долю джерелом, а іншу — стоком.
    3. Просто знайти потік в оригінальному графі.
    4. Звести до задачі комівояжера.
6. Що таке “мінімальний розріз” в контексті теореми Форда-Фалкерсона?
    1. Набір ребер з мінімальною вагою, видалення яких робить граф незв’язним.
    2. Поділ вершин на дві множини (S, T) так, що $s \in S, t \in T$, і сума пропускних здатностей ребер з S в T є мінімальною.
    3. Найкоротший шлях від джерела до стоку.
    4. Вершини з найменшою кількістю вхідних/вихідних ребер.
7. Якщо всі пропускні здатності в мережі збільшити в k разів, як зміниться величина максимального потоку?
    1. Збільшиться в k разів.
    2. Збільшиться в k^2 разів.
    3. Залишиться незмінною.
    4. Неможливо передбачити.
8. Що станеться, якщо в методі Форда-Фалкерсона на кожній ітерації вибирати доповнюючий шлях з найбільшою залишковою пропускною здатністю?

1. Це нічого не змінить.
2. Це гарантовано призведе до найменшої кількості ітерацій.
3. Це відома евристика, але вона не завжди оптимальна за кількістю ітерацій і складна в реалізації.
4. Алгоритм перестане бути коректним.

9. Якщо алгоритм Форда-Фалкерсона завершився, і ми розглянемо множину вершин S, досяжних з s в фінальній залишковій мережі, то розріз $(S, V \setminus S)$ буде:
    1. Завжди максимальним розрізом.
    2. Завжди мінімальним розрізом.
    3. Розрізом, що відповідає початковому потоку.
    4. Випадковим розрізом.
10. Чому для представлення потоку та пропускних здатностей часто використовують матриці, а не списки суміжності?
    1. Матриці завжди швидші.
    2. Списки суміжності не можуть представити орієнтований граф.
    3. Це спрощує доступ до пропускної здатності та потоку для будь-якої пари вершин, особливо для перевірки зворотних ребер.
    4. Це вимога самого методу.

Відповіді:

1. **3**. Едмондс-Карп — це конкретизація Форда-Фалкерсона, яка вимагає використовувати BFS для пошуку найкоротшого (за кількістю ребер) доповнюючого шляху.
2. **3**. Потік по зворотному ребру (v, u) в залишковій мережі означає, що ми "забираємо" потік з прямого ребра (u, v) для перенаправлення його іншим шляхом.
3. **3**. Якщо DFS буде систематично вибирати шлях $s \to u \to t$, він збільшить потік на 1. Потім, щоб використати шлях через v, йому

доведеться пустити потік по зворотному ребру $s \to v \to \cdots \to u \to s$ .... Неоптимальний вибір шляху через `u` може призвести до 2000 ітерацій по 1 одиниці потоку.

4. **2**. Це є ключовим результатом теореми. Відсутність доповнюючого шляху означає, що потік більше не можна збільшити.

5. **1**. Створюється супер-джерело s з ребрами до всіх вершин долі U і супер-стік t з ребрами від усіх вершин долі V. Пропускна здатність всіх ребер — 1. Величина макс. потоку дорівнює розміру макс. паросполучення.

6. **2**. Це формальне визначення розрізу. Мінімальний розріз — це розріз з найменшою можливою сумарною пропускною здатністю ребер, що його перетинають.

7. **1**. Якщо пропускна здатність кожного ребра множиться на k, то і пропускна здатність будь-якого розрізу множиться на k. Отже, і мінімальний розріз (а з ним і максимальний потік) збільшиться в k разів.

8. **3**. Ця “жадібна” евристика (відома як “fat-path heuristic”) може покращити продуктивність на деяких графах, але не гарантує оптимальної кількості ітерацій і її складно реалізувати ефективно.

9. **2**. Коли потік максимальний, в залишковій мережі немає шляху від s до t. Множина досяжних з s вершин S і недосяжних $V \setminus S$ утворює розріз, який і є мінімальним.

10. **3**. Матричне представлення `capacity[u][v]` і `flow[u][v]` дає $O(1)$ доступ до параметрів будь-якого ребра, що зручно для побудови залишкової мережі “на льоту”.

Завдання для самостійної роботи

1. Реалізуйте метод Форда-Фалкерсона, використовуючи пошук в ширину (BFS) для знаходження доповнюючих шляхів (це буде алгоритм Едмондса-Карпа).

2. Сконструюйте приклад мережі та послідовність вибору шляхів за допомогою DFS, яка призводить до великої кількості ітерацій, демонструючи недолік методу.

3. Напишіть програму, яка не лише знаходить максимальний потік, а й визначає мінімальний розріз мережі.
4. Реалізуйте розв'язання задачі про максимальне паросполучення в дводольному графі, звівши її до задачі про максимальний потік.
5. Дослідіть, як поводитиметься ваш код, якщо в мережі є кілька джерел або кілька стоків. Запропонуйте спосіб звести таку задачу до класичної.
6. Модифікуйте вашу реалізацію, щоб вона могла обробляти неорієнтовані ребра (кожне неорієнтоване ребро (u, v) з пропускною здатністю c можна замінити двома орієнтованими (u, v) і (v, u) з тією ж пропускною здатністю c).
7. Напишіть функцію, яка перевіряє, чи є заданий потік f у мережі G коректним (тобто задовольняє всім трьом умовам).
8. Реалізуйте задачу про максимальний потік з обмеженнями на пропускну здатність не лише ребер, а й вершин.
9. Уявіть, що ви можете збільшити пропускну здатність будь-якого одного ребра на k одиниць. Напишіть алгоритм, який визначає, яке ребро потрібно обрати, щоб отримати максимальний приріст загального потоку.
10. Візуалізуйте роботу методу Форда-Фалкерсона: на кожній ітерації показуйте оригінальну мережу з поточним потоком та залишкову мережу з знайденим доповнюючим шляхом.

Теми лабораторних робіт

1. Порівняльний аналіз продуктивності методу Форда-Фалкерсона з DFS та BFS (Едмондс-Карп) на різних типах мереж.
2. Розробка системи для розподілу ресурсів (наприклад, завдань між серверами) з використанням алгоритму максимального потоку.
3. Реалізація задачі про максимальне паросполучення та її застосування для системи рекомендацій "користувач-продукт".
4. Створення інтерактивного візуалізатора, що демонструє роботу методу Форда-Фалкерсона та концепцію залишкової мережі.

5. Задача про мультипотік: пошук максимального потоку між кількома парами джерел і стоків одночасно.
6. Застосування методу максимального потоку для аналізу вразливостей транспортних чи комунікаційних мереж (пошук мінімального розрізу).
7. Реалізація алгоритму максимального потоку для мереж з обмеженнями на пропускну здатність вершин.
8. Пошук максимального потоку мінімальної вартості (Min-cost max-flow problem).
9. Моделювання шляхів евакуації з будівлі за допомогою алгоритмів потоку.
10. Розробка модуля для гри в жанрі “tower defense”, де потік “ворогів” по шляхах оптимізується за допомогою потокових алгоритмів.
11. Реалізація методу “проштовхування-перепозначення” (Push-relabel) та порівняння його з методом Форда-Фалкерсона.
12. Аналіз зображень: реалізація алгоритму сегментації “Graph Cut” на основі мінімального розрізу.

Максимальний потік. Алгоритм Едмондса-Карпа

У попередньому розділі ми дослідили метод Форда-Фалкерсона — потужну, але загальну ідею ітеративного збільшення потоку. Ми також побачили його головний недолік: продуктивність методу критично залежить від того, як ми вибираємо доповнюючий шлях на кожній ітерації. Невдалий вибір може призвести до величезної кількості кроків, а для дійсних пропускних здатностей — навіть до нескінченної роботи.

Алгоритм Едмондса-Карпа — це не новий метод, а конкретизація методу Форда-Фалкерсона. Він пропонує просту, але надзвичайно ефективну стратегію вибору шляху, яка вирішує основні проблеми загального методу і гарантує поліноміальний час роботи. Ця стратегія полягає у виборі найкоротшого доповнюючого шляху.

Ідея алгоритму Едмондса-Карпа полягає в тому, щоб на кожному кроці методу Форда-Фалкерсона знаходити доповнюючий шлях, який має найменшу кількість ребер. Це “жадібне” рішення є дуже далекоглядним. Замість того, щоб потенційно вибрати довгий і “тонкий” шлях, ми завжди обираємо найкоротший, що дозволяє “насичувати” мережу більш рівномірно.

1.1. Пошук найкоротшого шляху за допомогою BFS

Ідеальним інструментом для пошуку найкоротшого шляху (за кількістю ребер) у незваженому графі є пошук в ширину (BFS). BFS досліджує граф “шар за шаром”, починаючи з джерела. Це гарантує, що коли ми вперше досягаємо будь-якої вершини (включаючи стік t), ми робимо це по шляху з мінімально можливою кількістю ребер.

Таким чином, алгоритм Едмондса-Карпа — це метод Форда-Фалкерсона, де для пошуку шляху в залишковій мережі на кожній ітерації використовується саме BFS.

Псевдокод алгоритму Едмондса-Карпа

Основний алгоритм залишається тим самим, що й у Форда-Фалкерсона, але виклик функції пошуку шляху конкретизовано.

```
function EdmondsKarp(G, s, t):
  // Ініціалізувати потік f нулем
  for each edge (u, v) in G.E:
    f(u, v) = 0

  max_flow = 0

  // Головний цикл
  while true:
    // 1. Знайти найкоротший доповнюючий шлях за допомогою BFS
    // BFS повертає словник батьківських вершин для відновлення шляху
    // і залишкову пропускну здатність цього шляху
    (parent_map, path_flow) = BFS_FindShortestPath(G, s, t, f)

    // 2. Якщо шлях не знайдено, завершуємо роботу
    if path_flow == 0:
      break

    // 3. Збільшуємо загальний потік
    max_flow = max_flow + path_flow
```

```
    // 4. Оновлюємо потік вздовж знайденого шляху
    v = t
    while v != s:
      u = parent_map[v]
      // Збільшуємо потік по прямому ребру або зменшуємо по зворотном
у
      f(u, v) = f(u, v) + path_flow
      f(v, u) = f(v, u) - path_flow // Антисиметрія
      v = u

  return max_flow
```

Псевдокод пошуку шляху за допомогою BFS

Ця функція є основою алгоритму. Вона повинна не лише знайти шлях, а й повернути інформацію для його відновлення та обчислення його пропускної здатності.

```
function BFS_FindShortestPath(G, s, t, f):
  // parent_map[v] = u означає, що в шляху ми прийшли у v з u
  parent_map = {s: NULL}

  // Створюємо чергу для BFS і додаємо джерело
  queue = new Queue()
  queue.enqueue(s)

  // Поки черга не порожня і стік не знайдено
  while not queue.is_empty():
    u = queue.dequeue()

    // Розглядаємо всіх сусідів v вершини u
    for each vertex v in G.V:
      // Обчислюємо залишкову пропускну здатність c(u,v) - f(u,v)
      residual_capacity = c(u, v) - f(u, v)

      // Якщо є вільне місце і вершина ще не відвідана
      if residual_capacity > 0 and v not in parent_map:
        parent_map[v] = u
        queue.enqueue(v)

        // Якщо досягли стоку, виходимо з циклу
        if v == t:
          // Спливаємо до кореневої умови
          goto path_found

path_found:
```

```
// Якщо стік недосяжний
if t not in parent_map:
  return (NULL, 0)

// Відновлюємо шлях і знаходимо його пропускну здатність
path_flow = ∞
v = t
while v != s:
  u = parent_map[v]
  // Шукаємо "вузьке місце"
  path_flow = min(path_flow, c(u, v) - f(u, v))
  v = u

return (parent_map, path_flow)
```

1.2. Аналіз складності

Ось де проявляється вся сила алгоритму Едмондса-Карпа. Його часова складність не залежить від величини пропускних здатностей і завжди є поліноміальною.

Доведення цього факту базується на ключовій властивості: відстань (кількість ребер) від джерела s до будь-якої вершини v у залишковій мережі, $\text{dist}_f(s, v)$, ніколи не зменшується з кожною ітерацією (аугментацією потоку).

Інтуїтивне пояснення: Коли ми знаходимо найкоротший шлях і насичуємо його “вузьке місце” (критичне ребро), це ребро зникає із залишкової мережі. Будь-який новий шлях до кінцевої вершини цього ребра тепер повинен буде йти “в обхід”, а отже, не може бути коротшим. Хоча в залишковій мережі з’являються зворотні ребра, можна довести, що вони не створюють коротших шляхів до будь-якої з вершин.

Формальний аналіз:

1. Кожна ітерація алгоритму включає запуск BFS, що займає $O(E)$ часу.
2. Можна довести, що будь-яке ребро (u, v) може стати “критичним” (тобто бути “вузьким місцем” на найкоротшому шляху) не більше ніж $\frac{|V|}{2}$ разів.
3. Оскільки всього ребер в графі може бути до $|E|$, загальна кількість ітерацій (аугментацій потоку) не перевищить $O(V \cdot E)$.

4. Перемножуючи час однієї ітерації на їхню максимальну кількість, отримуємо загальну часову складність:

$$O(E) \times O(V \cdot E) = O(V \cdot E^2)$$

Цей результат є величезним проривом у порівнянні з $O(|f_{max}| \cdot E)$ для загального методу Форда-Фалкерсона. Він гарантує, що задача буде розв'язана за розумний час навіть для мереж з великими пропускними здатностями.

Область застосування

Оскільки алгоритм Едмондса-Карпа є ефективною реалізацією методу пошуку максимального потоку, він застосовується у тих самих сферах, що й метод Форда-Фалкерсона, але з більшою впевненістю в продуктивності:

- Надійні транспортні та мережеві моделі: Де потрібен гарантований поліноміальний час розрахунку.
- Задача про максимальне паросполучення у дводольних графах: Це класичне застосування, де пропускні здатності дорівнюють 1, і алгоритм працює дуже швидко.
- Основа для вивчення складніших алгоритмів: Розуміння роботи Едмондса-Карпа та його аналізу складності є необхідним кроком перед вивченням швидших алгоритмів, таких як алгоритм Дініца.
- Задачі планування та розподілу ресурсів: Де потрібно знайти оптимальний розподіл, який можна змоделювати як потік.

Приклад застосування

Давайте повернемося до "проблемного" прикладу з попереднього розділу, де невдалий вибір шляху міг призвести до великої кількості ітерацій.

Мережа:

- Вершини: S, A, B, T
- Ребра та пропускні здатності: c(S,A)=1000, c(A,B)=1, c(B,T)=1000, c(S,B)=1000.

Уявімо, що у нас є також ребро c(A,T)=1000. Отже, мережа: c(S,A)=1000, c(S,B)=1000, c(A,B)=1, c(A,T)=1000, c(B,T)=1000.

Крок 0: Ініціалізація. Весь потік f дорівнює 0. $|f| = 0$.

Ітерація 1:

5. Пошук шляху (BFS):
    - BFS починає з S. Додає в чергу A і B.
    - З A досягає T. Шлях $S \to A \to T$ знайдено. Його довжина 2.
    - З B досягає T. Шлях $S \to B \to T$ знайдено. Його довжина 2.
    - BFS знайде один з цих шляхів (нехай $p_1 = S \to A \to T$). Шляхи довжиною 3 (як $S \to A \to B \to T$) будуть розглянуті пізніше і не будуть обрані, бо ми вже знайшли шлях до T.
6. Пропускна здатність шляху: $c_f(p_1) = \min(c(S,A), c(A,T)) = \min(1000,1000) = 1000$.
7. Збільшення потоку: $f(S,A) = 1000, f(A,T) = 1000$.
8. Результат: Загальний потік $|f| = 1000$.

Ітерація 2:

9. Пошук шляху (BFS) в залишковій мережі:
    - Ребра (S,A) і (A,T) насичені, їх немає в залишковій мережі (але є зворотні).
    - BFS з S знаходить шлях $p_2 = S \to B \to T$. Його довжина 2.
10. Пропускна здатність шляху: $c_f(p_2) = \min(c_f(S,B), c_f(B,T)) = \min(1000,1000) = 1000$.
11. Збільшення потоку: $f(S,B) = 1000, f(B,T) = 1000$.
12. Результат: Загальний потік $|f| = f(S,A) + f(S,B) = 1000 + 1000 = 2000$.

Ітерація 3:

13. Пошук шляху (BFS) в залишковій мережі:
    - Ребра (S,A), (A,T), (S,B), (B,T) насичені.
    - BFS починає з S. Не може досягти ні A, ні B (бо $c_f(S,A) = 0, c_f(S,B) = 0$).
    - BFS не може знайти шлях до T.

Завершення: Алгоритм зупиняється. Максимальний потік дорівнює 2000. Ми знайшли його всього за 2 ітерації. Загальний метод Форда-Фалкерсона з невдалим вибором шляху $S \to A \to B \to T$ потребував би 2000 ітерацій. Це яскраво демонструє перевагу вибору найкоротшого шляху.

Переваги та недоліки

Переваги

- Поліноміальна складність: Головна перевага. Час роботи $O(V \cdot E^2)$ не залежить від значень пропускних здатностей, що робить алгоритм надійним і передбачуваним.
- Гарантоване завершення: На відміну від загального методу, Едмондс-Карп гарантовано завершує роботу навіть для дійсних пропускних здатностей.
- Простота реалізації: Алгоритм є відносно простим для впровадження, оскільки базується на стандартному BFS, який є базовим графовим алгоритмом.

Недоліки

- Не найшвидший: Хоча $O(V \cdot E^2)$ — це добре, існують значно швидші алгоритми (наприклад, алгоритм Дініца зі складністю $O(V^2 \cdot E)$ або ще більш просунуті методи). Для великих та/або щільних графів Едмондс-Карп може бути занадто повільним.
- Кількість ітерацій: Хоча загальна кількість ітерацій обмежена ($O(V \cdot E)$), вона все ще може бути великою. Алгоритми, що можуть знаходити кілька доповнюючих шляхів за раз, працюють ефективніше.

Контрольні запитання

1. У чому полягає ключова відмінність алгоритму Едмондса-Карпа від загального методу Форда-Фалкерсона?
2. Чому для пошуку доповнюючого шляху використовується саме BFS, а не DFS?
3. Поясніть інтуїтивно, чому довжина найкоротшого доповнюючого шляху не може зменшуватися з кожною ітерацією.
4. Що обмежує загальну кількість ітерацій в алгоритмі Едмондса-Карпа?

5. Порівняйте складність $O(V \cdot E^2)$ зі складністю $O(|f_{max}| \cdot E)$. В яких випадках Едмондс-Карп буде значно кращим?

6. Чи зміниться результат роботи алгоритму, якщо в мережі є кілька різних найкоротших шляхів однакової довжини?

7. Чи є алгоритм Едмондса-Карпа “жадібним”? Якщо так, то в чому полягає його “жадібний” вибір?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Едмондса-Карпа для пошуку максимального потоку.

2. Мета

– Поглибити розуміння методу Форда-Фалкерсона шляхом реалізації його ефективної версії.

– Набути практичних навичок реалізації алгоритму пошуку в ширину (BFS) в контексті задачі про потік.

– Проаналізувати та порівняти продуктивність конкретизованого алгоритму з загальним методом.

3. Завдання

a) Спроектувати та реалізувати структуру даних для потокової мережі.

b) Реалізувати функцію пошуку найкоротшого доповнюючого шляху за допомогою BFS. Функція повинна вміти працювати з залишковою мережею.

c) На основі цієї функції реалізувати основний цикл алгоритму Едмондса-Карпа.

d) Програма повинна читати опис мережі з файлу та виводити величину максимального потоку.

e) (Додатково) Реалізувати лічильник ітерацій та порівняти кількість ітерацій для вашої попередньої реалізації (з DFS) та поточної (з BFS) на спеціально підібраних тестових випадках.

4. Опис

Лабораторна робота є прямим продовженням попередньої. Замість використання DFS для пошуку довільного шляху, студент повинен реалізувати пошук саме найкоротшого шляху за допомогою BFS, тим самим перетворивши свою програму на реалізацію алгоритму Едмондса-Карпа. Головний акцент — на коректній реалізації BFS для залишкової мережі та розумінні переваг такого підходу.

Порядок виконання

a) Адаптація коду: Візьміть за основу код з попередньої лабораторної роботи. Основні структури даних для графа, потоку та пропускних здатностей залишаться тими ж.
b) Реалізація BFS: Напишіть нову функцію `BFS_FindShortestPath`, яка замінить `FindAugmentingPath_DFS`. Вона повинна використовувати чергу (`Queue`) для зберігання вершин до обробки та словник (`map` або масив) для зберігання батьківських вершин (`parent_map`), щоб відновити шлях.
c) Інтеграція BFS: Замініть виклик старої функції пошуку шляху на нову у вашому головному циклі.
d) Оновлення логіки: Переконайтеся, що логіка обчислення залишкової пропускної здатності шляху та оновлення потоку працює коректно з даними, які повертає ваша BFS-функція.
e) Тестування: Використовуйте ті ж самі тестові випадки, що й раніше. Особливу увагу приділіть графам, на яких DFS працював повільно. Переконайтесь, що BFS знаходить потік за значно меншу кількість ітерацій.
f) Аналіз: Проаналізуйте отримані результати, порівняйте кількість ітерацій та зробіть висновки про ефективність алгоритму Едмондса-Карпа.

5. Результат

Результатом роботи є програма, що реалізує алгоритм Едмондса-Карпа, та звіт, що містить опис реалізації, результати тестування та порівняльний аналіз з менш ефективним підходом.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

   – Реалізація алгоритму пошуку в ширину (BFS) для знаходження найкоротшого шляху в залишковій мережі.

- Основний цикл, що використовує BFS.

2. Продемонструвати роботу додатку:
   - На тестовому прикладі, де вибір найкоротшого шляху є критичним для швидкої збіжності.
   - Порівняти (якщо можливо) кількість ітерацій з реалізацією на основі DFS.
3. Бути готовим відповісти на питання:
   - Питання з розділу “Контрольні запитання”.
   - Які структури даних ви використали для реалізації черги в BFS?
   - Поясніть на прикладі, як ваша BFS-функція відновлює найкоротший шлях після того, як досягла стоку.
   - Чому часова складність вашої реалізації є поліноміальною?
   - Чи є випадки, коли реалізація на основі DFS може відпрацювати швидше (за реальним часом), ніж на основі BFS?

Тести

1. Яку основну модифікацію вносить алгоритм Едмондса-Карпа в метод Форда-Фалкерсона?
   1. Використовує матрицю суміжності замість списків.
   2. Шукає доповнюючий шлях з максимальною пропускною здатністю.
   3. Шукає найкоротший доповнюючий шлях (за кількістю ребер).
   4. Працює зі зваженими графами.
2. Який стандартний алгоритм використовується для пошуку шляху в алгоритмі Едмондса-Карпа?
   1. Пошук в глибину (DFS).
   2. Пошук в ширину (BFS).
   3. Алгоритм Дейкстри.

4. Алгоритм Флойда-Воршелла.

3. Яка часова складність алгоритму Едмондса-Карпа?

    1. $O(|f_{max}| \cdot E)$
    2. $O(V \cdot E^2)$
    3. $O(E\log V)$
    4. $O(V^3)$

4. Чому складність алгоритму Едмондса-Карпа є значним покращенням у порівнянні з загальним методом Форда-Фалкерсона?

    1. Вона простіша для обчислення.
    2. Вона не залежить від значень пропускних здатностей.
    3. Вона краще працює на щільних графах.
    4. Вона лінійна.

5. Ключова властивість, що лежить в основі доведення складності Едмондса-Карпа, полягає в тому, що:

    1. Величина потоку на кожному кроці зростає експоненційно.
    2. Довжина найкоротшого доповнюючого шляху ніколи не зменшується.
    3. Граф стає розрідженим після кількох ітерацій.
    4. BFS працює за константний час.

Відповіді: 1. - 3), 2. - 2), 3. - 2), 4. - 2), 5. - 2)

Тести підвищеної складності

1. Скільки разів, згідно з доведенням, одне й те саме ребро може стати “критичним” в алгоритмі Едмондса-Карпа?

    1. Не більше одного разу.
    2. $O(E)$ разів.
    3. $O(V)$ разів.

4. $O(\log V)$ разів.

2. Для якого типу графів складність Едмондса-Карпа $O(V \cdot E^2)$ є найгіршою?
    1. Для розріджених графів (де $|E| \approx |V|$).
    2. Для повних щільних графів (де $|E| \approx |V|^2|$).
    3. Для дерев.
    4. Для дводольних графів.
3. Що можна сказати про кількість ітерацій алгоритму Едмондса-Карпа?
    1. Вона обмежена величиною $|f_{max}|$.
    2. Вона обмежена добутком $O(V \cdot E)$.
    3. Вона обмежена $O(V + E)$.
    4. Вона не обмежена.
4. Якщо мережа є одиничною (всі пропускні здатності дорівнюють 1), якою буде складність алгоритму Едмондса-Карпа?
    1. $O(V \cdot E^2)$
    2. $O(E\sqrt{V})$
    3. $O(V + E)$
    4. $O(\min(V^{2/3}, E^{1/2}) \cdot E)$
5. Чи впливає на коректність алгоритму Едмондса-Карпа наявність циклів в графі?
    1. Так, алгоритм може зациклитися.
    2. Ні, BFS і робота з залишковою мережею коректно обробляють цикли.
    3. Так, якщо цикл проходить через джерело або стік.
    4. Тільки якщо цикли містять ребра з нульовою пропускною здатністю.

6. У порівнянні з алгоритмом Дініца, алгоритм Едмондса-Карпа:
    1. Завжди швидший.
    2. Має кращу асимптотичну складність.
    3. Простіший в реалізації, але зазвичай повільніший.
    4. Використовує іншу основну ідею.
7. Яка головна причина того, що довжина найкоротшого шляху $\text{dist}_f(s, v)$ не зменшується?
    1. BFS завжди знаходить один і той же шлях.
    2. Зникнення ребер з залишкової мережі може лише збільшити або не змінити відстані.
    3. Кількість вершин в графі постійна.
    4. Потік завжди зростає.
8. Чи знайде Едмондс-Карп той самий кінцевий потік (тобто ті ж значення $f(u, v)$ для кожного ребра), що й Форд-Фалкерсон з DFS?
    1. Завжди знайде той самий потік.
    2. Завжди знайде інший потік.
    3. Величина максимального потоку буде та сама, але розподіл потоку по ребрах може відрізнятися.
    4. Неможливо передбачити.
9. Якщо граф є орієнтованим ациклічним графом (DAG), чи дає це якусь перевагу для алгоритму Едмондса-Карпа?
    1. Так, його складність покращується до $O(V \cdot E)$.
    2. Ні, наявність чи відсутність циклів не впливає на загальну складність.
    3. Так, можна використовувати топологічне сортування замість BFS.
    4. Ні, але це спрощує реалізацію зворотних ребер.

10. Чому алгоритм Едмондса-Карпа вважається “хорошим” алгоритмом в теоретичному сенсі?

    1. Тому що він найшвидший на практиці.
    2. Тому що його часова складність є поліноміальною від розміру вхідних даних ($|V|$ і $|E|$).
    3. Тому що він потребує найменше пам’яті.
    4. Тому що його написали відомі вчені.

Відповіді:

1. 3). Доводиться, що відстань від джерела до "початкової" вершини критичного ребра збільшується кожного разу, коли це ребро стає критичним. Оскільки відстань не може перевищувати $|V|$, ребро може бути критичним $O(V)$ разів.

2. 2). Для щільних графів, де $|E| = O(|V|^2)$, складність стає $O(|V| \cdot (|V|^2)^2) = O(|V|^5)$. Це найгірший сценарій.

3. 2). Це випливає з аналізу складності: кожна з $O(V \cdot E)$ аугментацій вимагає одного запуску BFS.

4. 4). Для одиничних мереж існують значно ефективніші алгоритми. Один з них (алгоритм Хопкрофта-Карпа для дводольних графів, який є частковим випадком) має складність $O(E\sqrt{V})$, а алгоритм Дініца дає $O(\min(V^{2/3}, E^{1/2}) \cdot E)$. Складність самого Едмондса-Карпа не змінюється, але це питання на знання більш просунутих результатів.

5. 2). BFS за своєю природою коректно працює з будь-якими графами, включаючи ті, що містять цикли, оскільки він відстежує відвідані вершини і ніколи не заходить в одну вершину двічі в рамках одного пошуку.

6. 3). Алгоритм Дініца має кращу асимптотичну складність ($O(V^2E)$) і на практиці значно швидший, але його концепція "блокуючого потоку" і "шаруватої мережі" є складнішою для розуміння та реалізації.

7. 2). Коли критичне ребро (u, v) зникає з залишкової мережі, будь-який новий шлях до v мусить іти "в обхід", що не може зробити його коротшим.

8. 3). Величина потоку буде однаковою (оскільки потік максимальний і унікальний за величиною), але шляхи, якими цей потік розподілений по ребрах, можуть бути різними залежно від послідовності аугментацій.

9. 2). Ациклічність графа не спрощує завдання для Едмондса-Карпа, оскільки залишкова мережа все одно містить зворотні ребра, які можуть утворювати цикли.

10. 2). В теорії алгоритмів "хорошими" або "ефективними" вважаються ті, чий час роботи обмежений поліномом від розміру вхідних даних (кількості вершин і ребер), а не від значень цих даних (як-от пропускних здатностей).

Завдання для самостійної роботи

1. Реалізуйте лічильник, який підраховує, скільки разів кожне ребро ставало “критичним” в ході роботи алгоритму Едмондса-Карпа. Перевірте теоретичне обмеження ($O(V)$) на практиці.
2. Напишіть програму, яка генерує випадкову потокову мережу із заданими параметрами (кількість вершин, щільність, діапазон пропускних здатностей).
3. Порівняйте реальний час роботи вашої реалізації Едмондса-Карпа на випадкових щільних та розріджених графах. Побудуйте графік залежності часу від розміру графа.
4. Дослідіть, як впливає на кількість ітерацій наявність ребер з дуже великою пропускною здатністю порівняно з ребрами з малою пропускною здатністю.
5. Реалізуйте задачу “перевірка потоку”: напишіть функцію, яка приймає мережу та потік, і перевіряє, чи є цей потік максимальним.
6. Спробуйте оптимізувати вашу реалізацію BFS, наприклад, припиняючи пошук на певному шарі, якщо стало відомо, що стік недосяжний.
7. Задача про декілька джерел та стоків: модифікуйте мережу (додавши “супер-джерело” та “супер-стік”) для розв’язання цієї задачі за допомогою стандартного алгоритму Едмондса-Карпа.

8. Напишіть програму, що знаходить не один, а всі можливі мінімальні розрізи в мережі.
9. Дослідіть, як зміниться максимальний потік, якщо видалити з мережі одну вершину (разом з усіма інцидентними їй ребрами).
10. Реалізуйте алгоритм, що знаходить доповнюючий шлях з максимальною пропускною здатністю (не найкоротший) і порівняйте його ефективність з Едмондсом-Карпом.

Теми лабораторних робіт

1. Порівняльний аналіз продуктивності алгоритмів Форда-Фалкерсона (з DFS) та Едмондса-Карпа.
2. Застосування алгоритму Едмондса-Карпа для моделювання та аналізу транспортних потоків у міській мережі.
3. Розробка системи для знаходження “вузьких місць” в комп’ютерній мережі на основі мінімального розрізу, знайденого за допомогою Едмондса-Карпа.
4. Візуалізатор роботи алгоритму Едмондса-Карпа, що показує стан черги BFS та побудову найкоротшого шляху на кожній ітерації.
5. Реалізація задачі про максимальне паросполучення у дводольному графі з використанням алгоритму Едмондса-Карпа.
6. Задача про циркуляцію з вимогами: розширення моделі потоку, де для вершин є вимоги щодо вхідного/вихідного потоку.
7. Аналіз надійності мережі: пошук реберної та вершинної зв’язності графа за допомогою багаторазових запусків алгоритму максимального потоку.
8. Моделювання фінансових потоків між організаціями з лімітами на транзакції.
9. Оптимізація розкладу: зведення задачі про складання розкладу до задачі про потік.
10. Реалізація алгоритму Едмондса-Карпа для розріджених графів з використанням списків суміжності.

11. Порівняння продуктивності Едмондса-Карпа та алгоритму Дініца (якщо він буде реалізований).

12. Розробка спрощеної системи сегментації зображень на основі мінімального розрізу.

Максимальний потік. Алгоритм Дініца

Алгоритм Едмондса-Карпа був значним покращенням, оскільки його складність $O(V \cdot E^2)$ не залежала від величини пропускних здатностей. Проте він все ще може бути неефективним. Його головний недолік полягає в тому, що на кожній ітерації він виконує повний пошук в ширину (BFS) лише для того, щоб знайти один доповнюючий шлях.

Алгоритм Дініца пропонує набагато потужніший підхід. Замість того, щоб шукати шляхи поодинці, він використовує результати одного запуску BFS, щоб знайти цілий набір доповнюючих шляхів одночасно. Це досягається завдяки двом ключовим концепціям: допоміжній рівневому графі та блокуючому потоку.

1. Теоретичні відомості

Алгоритм Дініца, як і його попередники, працює ітеративно. Проте його ітерації, що називаються Крок ми, є значно складнішими та продуктивнішими. У кожній фазі алгоритм робить наступне:

1. Будує допоміжну шарувату мережу на основі поточної залишкової мережі.

2. Знаходить так званий блокуючий потік у цій шаруватій мережі.

3. Додає знайдений блокуючий потік до загального потоку. Фази повторюються, доки стік залишається досяжним з джерела.

1.1. Допоміжний рівневий граф

Це перша ключова ідея алгоритму. Рівневий граф L — це ациклічний підграф залишкової мережі G_f, який містить лише ті ребра, що ведуть до “наступного шару” на найкоротшому шляху від джерела s.

Побудова шаруватої мережі:

4. Виконуємо пошук в ширину (BFS) з вершини s у залишковій мережі G_f. Мета цього BFS — обчислити для кожної вершини v її рівень

(level) або відстань $\text{dist}(v)$ — довжину найкоротшого шляху (в кількості ребер) від s до v.

5. Якщо під час BFS стік t виявився недосяжним, це означає, що доповнюючих шляхів більше не існує. Алгоритм завершується.

6. Якщо стік t досяжний, рівневий граф L складається з усіх вершин V і лише тих ребер (u, v) із залишкової мережі G_f, для яких виконується умова:

$$\text{dist}(v) = \text{dist}(u) + 1$$

Це означає, що ми залишаємо тільки ті ребра, які ведуть нас строго "вперед" по найкоротших шляхах до стоку. Всі ребра, що ведуть "назад" або між вершинами одного рівня, ігноруються. Завдяки цьому рівневий граф завжди є орієнтованим ациклічним графом (DAG).

Псевдокод побудови рівневого графу такий (за допомогою BFS)

```
function BuildLevelGraph_BFS(G_f, s, t, dist):
  // dist - масив для зберігання відстаней/рівнів
  for each vertex v in G.V:
    dist[v] = -1 // -1 означає, що вершина не відвідана

  dist[s] = 0
  queue = new Queue()
  queue.enqueue(s)

  while not queue.is_empty():
    u = queue.dequeue()

    // Розглядаємо всіх сусідів v вершини u в залишковій мережі
    for each neighbor v of u in G_f:
      if dist[v] < 0 and G_f.capacity(u, v) > 0:
        dist[v] = dist[u] + 1
        queue.enqueue(v)

  // Повертаємо true, якщо стік досяжний, інакше false
  return dist[t] != -1
```

1.2. Блокуючий потік (Blocking Flow)

Це друга ключова ідея. Побудувавши рівневий граф L, ми не шукаємо в ній один шлях. Натомість ми намагаємося "протиснути" через неї якомога більше потоку, доки вона не стане "заблокованою".

Блокуючий потік — це такий потік f' у рівневому графі L, що на кожному шляху від s до t в L є принаймні одне ребро, насичене цим потоком f' (тобто для якого потік дорівнює пропускній здатності).

Для пошуку блокуючого потоку використовується пошук в глибину (DFS). DFS запускається з джерела s і шукає будь-який шлях до стоку t, рухаючись лише по ребрах шаруватої мережі.

- Коли DFS знаходить шлях, він визначає його пропускну здатність, збільшує потік і повертається назад.
- На відміну від простого пошуку, DFS в алгоритмі Дініца не завершується після знаходження одного шляху. Він продовжує пошук з поточної вершини, намагаючись знайти інші шляхи, доки не будуть досліджені всі можливості.

Оптимізація DFS: Для уникнення повторного відвідування “тупикових” гілок використовується оптимізація з вказівниками. Для кожної вершини ми зберігаємо вказівник на останнє ребро, яке ми розглядали. Якщо DFS з вершини u повертається, не знайшовши шлях до стоку, ми знаємо, що з цієї вершини шляху більше немає в поточній фазі, і більше не заходимо в неї.

1.3. Повний алгоритм Дініца

Алгоритм складається з послідовності фаз.

Псевдокод алгоритму Дініца

```
function Dinic(G, s, t):
  max_flow = 0
  dist = new array(|V|) // Масив для рівнів

  // Поки BFS може знайти шлях до стоку
  while BuildLevelGraph_BFS(G_f, s, t, dist):
    // ptr - масив вказівників для DFS оптимізації
    ptr = new array(|V|)

    // Шукаємо блокуючий потік і додаємо його до загального
    while pushed_flow = FindBlockingFlow_DFS(s, t, ∞, G_f, dist, ptr)
:
      max_flow = max_flow + pushed_flow

  return max_flow
```

Псевдокод пошуку блокуючого потоку (за допомогою DFS)

Це найскладніша частина. Функція є рекурсивною.

```
function FindBlockingFlow_DFS(u, t, pushed, G_f, dist, ptr):
  // Базовий випадок: досягли стоку
  if pushed == 0: return 0
  if u == t: return pushed

  // Ітеруємо по сусідах вершини u, починаючи з останнього вказівника
  for v from ptr[u] to num_neighbors(u):
    // Перевіряємо, що ребро веде на наступний рівень
    if dist[v] != dist[u] + 1:
      continue // Пропускаємо ребро

    residual_capacity = G_f.capacity(u, v)

    // Намагаємося проштовхнути потік
    tr = FindBlockingFlow_DFS(v, t, min(pushed, residual_capacity), G
_f, dist, ptr)

    if tr == 0: // Якщо потік не вдалося проштовхнути
      ptr[u]++ // Переміщуємо вказівник на наступне ребро
      continue

    // Потік успішно пройшов, оновлюємо залишкову мережу "на льоту"
    G_f.decrease_capacity(u, v, tr) // f(u,v) += tr
    G_f.increase_capacity(v, u, tr) // f(v,u) -= tr

    return tr // Повертаємо обсяг проштовхнутого потоку

  return 0 // Якщо з вершини u не вдалося знайти шлях
```

1.4. Аналіз складності

Складність алгоритму Дініца є його головною перевагою.

7. Кількість фаз: Довжина найкоротшого шляху від s до t в залишковій мережі строго зростає з кожною фазою. Оскільки найдовший можливий простий шлях має довжину $|V| - 1$, кількість фаз не перевищує $|V| - 1$, тобто $O(V)$.

8. Складність однієї фази:

   – Побудова рівневого графу (BFS) займає $O(E)$ часу.

   – Пошук блокуючого потоку за допомогою DFS з оптимізацією вказівників займає $O(V \cdot E)$ часу. Це тому, що кожен раз, коли ми просуваємо потік, ми насичуємо принаймні одне ребро, а

всього таких насичень може бути $O(E)$. Пошук кожного шляху займає $O(V)$.

9. Загальна складність:

(Кількість фаз) × (Складність фази) $= O(V) \times O(V \cdot E) = O(V^2 \cdot E)$

Це значно краще, ніж $O(V \cdot E^2)$ для алгоритму Едмондса-Карпа, особливо на розріджених графах.

– Особливі випадки:

  – Для одиничних мереж (де всі пропускні здатності дорівнюють 1), наприклад, у задачі про максимальне паросполучення, складність алгоритму Дініца становить $O(E\sqrt{V})$.

  – Для планарних графів складність ще краща: $O(V\log V)$.

Область застосування

Алгоритм Дініца є одним з найшвидших та найпопулярніших алгоритмів для розв'язання задачі про максимальний потік на практиці.

– Конкурентне програмування: Це де-факто стандарт для задач на максимальний потік завдяки його високій продуктивності.

– Високопродуктивні обчислення: У задачах, де швидкість має критичне значення (наприклад, аналіз великих мереж, біоінформатика), алгоритм Дініца є кращим вибором, ніж Едмондс-Карп.

– Складні оптимізаційні задачі: Багато складніших задач (наприклад, потік мінімальної вартості) використовують алгоритми пошуку максимального потоку як свою складову, і швидкість цього компонента є дуже важливою.

Приклад застосування

Візьмемо ту ж мережу, що і в попередніх розділах: c(S,A)=10, c(S,B)=10, c(A,B)=2, c(A,T)=10, c(B,T)=10.

Крок 1

1. Побудова рівневого графу:

  – Запускаємо BFS з S.

- dist(S) = 0.
- dist(A) = 1, dist(B) = 1.
- dist(T) = 2 (досяжний і через A, і через B).
- Рівневий граф L_1 міститиме ребра (S,A), (S,B), (A,T), (B,T). Ребро (A,B) не увійде, бо dist(B) не дорівнює dist(A)+1.

2. Пошук блокуючого потоку в L_1:
   - Запускаємо DFS з S. Нехай він знаходить шлях $S \to A \to T$. Пропускна здатність $\min(10,10) = 10$. Збільшуємо потік на 10 по цьому шляху.
   - DFS повертається в S. Він не завершується, а продовжує пошук!
   - Він знаходить інший шлях $S \to B \to T$. Пропускна здатність $\min(10,10) = 10$. Збільшуємо потік на 10 по цьому шляху.
   - Тепер з S неможливо почати новий шлях до T в L_1, бо всі вихідні ребра ведуть до вершин A і B, з яких вихідні ребра (A,T) і (B,T) вже насичені потоком, знайденим у цій же фазі.
   - Блокуючий потік знайдено. Його величина 10 + 10 = 20.
3. Результат Фази 1: Загальний потік $|f| = 20$.

Крок 2

4. Побудова рівневого графу:
   - Будуємо нову залишкову мережу. Ребра (S,A), (S,B), (A,T), (B,T) насичені, але з'явились зворотні ребра.
   - Запускаємо BFS з S. Він не може досягти ні A, ні B. Стік T виявляється недосяжним.
5. Завершення: BFS повернув `false`. Алгоритм завершується.

Результат: Максимальний потік дорівнює 20. Алгоритм Дініца знайшов його за одну фазу, виконавши один BFS і один "великий" DFS, що знайшов два шляхи. Це демонструє його ефективність.

Переваги та недоліки

Переваги

– Висока продуктивність: Асимптотична складність $O(V^2 \cdot E)$ є однією з найкращих серед поліноміальних алгоритмів і значно перевершує Едмондса-Карпа.

– Дуже швидкий на практиці: Завдяки роботі з блокуючими потоками, алгоритм часто працює набагато швидше, ніж показує його теоретична оцінка в найгіршому випадку.

– Надзвичайна ефективність на одиничних мережах: Складність $O(E\sqrt{V})$ робить його ідеальним для задачі про максимальне паросполучення.

Недоліки

– Складність реалізації: Алгоритм значно складніший для розуміння та коректного впровадження, ніж Едмондс-Карп. Концепції шаруватої мережі, блокуючого потоку та оптимізованого DFS вимагають уважності.

– Більший обсяг коду: Реалізація потребує більше коду та складніших структур даних, що підвищує ймовірність помилок.

Контрольні запитання

1. У чому головна ідея, що відрізняє алгоритм Дініца від алгоритму Едмондса-Карпа?
2. Що таке “шарувата мережа” і яку властивість вона завжди має?
3. Як будується шарувата мережа? Який алгоритм для цього використовується?
4. Що таке “блокуючий потік”?
5. Який алгоритм використовується для пошуку блокуючого потоку і чому він ефективніший за простий пошук одного шляху?
6. З чого складається одна “Крок ” алгоритму Дініца?
7. Скільки щонайбільше фаз може бути в алгоритмі Дініца і чому?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Дініца для пошуку максимального потоку.

2. Мета

- Опанувати один з найефективніших класичних алгоритмів пошуку максимального потоку.
- Навчитися реалізовувати складні комбіновані алгоритми (що використовують і BFS, і DFS).
- Отримати практичні навички роботи з концепціями рівневого графу та блокуючого потоку.

3. Завдання

- Спроектувати та реалізувати структуру даних для потокової мережі.
- Реалізувати функцію `BuildLevelGraph_BFS` для побудови рівневого графу (обчислення рівнів).
- Реалізувати рекурсивну функцію `FindBlockingFlow_DFS` для пошуку блокуючого потоку в шаруватій мережі.
- Об'єднати ці функції в головному циклі алгоритму Дініца.
- Програма повинна читати опис мережі з файлу та виводити величину максимального потоку.

4. Опис

Ця лабораторна робота є значним кроком уперед за складністю. Студенту потрібно реалізувати двокомпонентну логіку фаз: спочатку побудувати допоміжну структуру (шарувату мережу) за допомогою BFS, а потім ефективно знайти в ній потік за допомогою DFS. Основний виклик полягає в коректній взаємодії цих двох частин.

5. Порядок виконання

- Проектування: Ретельно сплануйте структуру програми. Розділіть її на логічні блоки: представлення графа, функція BFS, функція DFS, головна функція.
- Реалізація BFS: Напишіть функцію, що обчислює масив відстаней `dist` від джерела. Вона повинна повертати `true` або `false` залежно від досяжності стоку.
- Реалізація DFS: Це найскладніший етап. Реалізуйте рекурсивну функцію, яка приймає поточну вершину, обсяг потоку, який можна

“проштовхнути”, та посилання на структури даних (граф, масив рівнів). Рух дозволяється лише до вершини наступного рівня.

- Реалізація головного циклу: Напишіть цикл `while`, який викликає BFS. Якщо стік досяжний, всередині запускається інший цикл `while`, який викликає DFS доти, доки не буде знайдено блокуючий потік.
- Тестування: Створіть набір тестів, що перевіряють різні сценарії: прості мережі, мережі з “довгими” та “короткими” шляхами, великі мережі. Порівняйте продуктивність з реалізацією алгоритму Едмондса-Карпа.

6. Результат

Результатом роботи є програма, що реалізує алгоритм Дініца, та звіт, що детально описує складні аспекти реалізації та демонструє переваги алгоритму над простішими аналогами.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
  - Функція побудови рівневого графу (BFS).
  - Функція пошуку блокуючого потоку (DFS).
  - Основний цикл, що керує Крок ми алгоритму.
- Продемонструвати роботу додатку:
  - На кількох тестових прикладах.
  - Продемонструвати покроково (за Крок ми) роботу алгоритму на невеликому прикладі, пояснюючи, як будується рівневий граф і як знаходиться блокуючий потік.
- Бути готовим відповісти на питання:
  - Питання з розділу “Контрольні запитання”.
  - Поясніть, як працює оптимізація з вказівниками у вашій реалізації DFS.
  - Чому рівневий граф гарантовано є ациклічним графом?

– У чому різниця між потоком, знайденим за одну ітерацію Едмондса-Карпа, і блокуючим потоком, знайденим за одну фазу Дініца?

Тести

1. Основна ідея алгоритму Дініца полягає у:

a) Пошуку одного найкоротшого шляху на кожній ітерації

b) Пошуку одного шляху з максимальною пропускною здатністю

c) Пошуку блокуючого потоку в шаруватій мережі

d) Пошуку потоку в зворотному напрямку

2. Рівневий граф в алгоритмі Дініца будується за допомогою:

a) Пошуку в глибину (DFS)

b) Пошуку в ширину (BFS)

c) Алгоритму Дейкстри

d) Топологічного сортування

3. Яку властивість завжди має рівневий граф?

a) Він є повним графом

b) Він є неорієнтованим

c) Він є дводольним

d) Він є ациклічним

5. Загальна часова складність алгоритму Дініца:

a) $O(V \cdot E^2)$

b) $O(V^2 \cdot E)$

c) $O(E \log V)$

d) $O(|f_{max}| \cdot E)$

7. Для пошуку блокуючого потоку в рівневому графі використовується:

a) Пошук в ширину (BFS)

b) Пошук в глибину (DFS)

c) Жадібний алгоритм

d) Динамічне програмування

Відповіді: 1. - c), 2. - b), 3. - d, 4. - b), 5. - b).

Тести підвищеної складності

1. Чому кількість фаз в алгоритмі Дініца не перевищує $|V| - 1$?

a) Тому що на кожній фазі до потоку додається щонайменше 1.

b) Тому що довжина найкоротшого шляху до стоку строго зростає з кожною фазою.

c) Тому що DFS відвідує кожну вершину не більше одного разу.

d) Це емпіричне спостереження, а не теоретичний факт.

2. Якою є складність алгоритму Дініца для задачі про максимальне паросполучення у дводольному графі?

a) $O(V^2 \cdot E)$

b) $O(E \log V)$

c) $O(E\sqrt{V})$

d) $O(V + E)$

3. Що таке “блокуючий потік”?

a) Потік, що дорівнює максимальному потоку в мережі.

b) Потік, що насичує принаймні одне ребро на кожному $s - t$ шляху в шаруватій мережі.

c) Потік, що проходить лише по одному шляху.

d) Потік, який неможливо збільшити.

4. Оптимізація з вказівниками в DFS-частині алгоритму Дініца служить для:

a) Прискорення доступу до сусідніх вершин.

b) Зменшення використання пам’яті.

c) Уникнення повторного обходу “тупикових” гілок в межах однієї фази.

d) Вибору ребра з найбільшою залишковою пропускною здатністю.

5. У порівнянні з алгоритмом “проштовхування-перепозначення”, алгоритм Дініца:

a) Завжди асимптотично швидший.

b) Має таку ж теоретичну складність, але іншу ідею.

c) Зазвичай повільніший на практиці, але простіший в реалізації.

d) Простіший в реалізації і часто конкурентоспроможний за швидкістю.

6. Якщо після побудови рівневого графу виявилось, що стік t не належить до неї (недосяжний), це означає, що:

a) В алгоритмі сталася помилка.

b) Потрібно запустити BFS ще раз.

c) Максимальний потік вже знайдено.

d) Потрібно шукати блокуючий потік в усій залишковій мережі.

7. У чому головна відмінність “фази” Дініца від “ітерації” Едмондса-Карпа?

a) Крок Дініца може знайти кілька доповнюючих шляхів, а ітерація Едмондса-Карпа — лише один.

b) Крок Дініца працює довше.

c) Крок Дініца використовує DFS, а ітерація Едмондса-Карпа — BFS.

d) Принципової відмінності немає.

8. Що станеться, якщо для пошуку блокуючого потоку замість DFS використати BFS?

a) Алгоритм працюватиме швидше.

b) Алгоритм перестане бути коректним.

c) Алгоритм знайде лише один шлях, і ефективність буде втрачена.

d) Нічого не зміниться.

9. Чи можна реалізувати алгоритм Дініца без явного створення шаруватої мережі?

a) Ні, це обов'язковий крок.

b) Так, можна передавати масив рівнів `dist` в DFS і перевіряти умову `dist[v] == dist[u] + 1` на льоту.

c) Так, але це сильно уповільнить алгоритм.

d) Ні, бо DFS не може працювати без ациклічного графа.

10. "Золотим стандартом" для вирішення задачі максимального потоку в конкурентному програмуванні вважається:

a) Едмондс-Карп через простоту.

b) Форд-Фалкерсон через гнучкість.

c) Алгоритм Дініца через високу практичну продуктивність.

d) Алгоритм "проштовхування-перепозначення" через найкращу теоретичну складність.

Відповіді на тести підвищеної складності

1. b). Довжина найкоротшого шляху до стоку строго зростає з кожною фазою, а максимальна довжина простого шляху обмежена $|V| - 1$.

2. c). Для одиничних мереж, до яких зводиться задача про паросполучення, складність Дініца значно покращується до $O(E\sqrt{V})$.

3. b). Це потік, який "блокує" подальший пошук шляхів у поточній шаруватій мережі, насичуючи хоча б одне ребро на кожному $s - t$ шляху.

4. c). Ця оптимізація дозволяє не розглядати повторно вершини, з яких вже неможливо досягти стоку в межах поточної фази.

5. d). Хоча Push-Relabel може мати кращу теоретичну складність у найгіршому випадку, на практиці добре реалізований алгоритм Дініца часто є швидшим і завжди простішим для написання.

6. c). Якщо BFS не може досягти стоку, це означає, що в залишковій мережі немає доповнюючих шляхів, а отже, знайдений потік є максимальним.

7. a). Ітерація Едмондса-Карпа знаходить один шлях. Крок Дініца використовує один BFS для побудови шаруватої мережі, а потім знаходить у ній потенційно багато шляхів за допомогою DFS.

8. c). BFS за своєю природою знаходить лише один шлях за один запуск. Використання його для пошуку блокуючого потоку звело б алгоритм Дініца до Едмондса-Карпа.

9. b). Явне створення окремої структури даних для шаруватої мережі не є обов'язковим. Ефективніше передавати масив рівнів у DFS і перевіряти умову на льоту.

10. c). Через чудовий баланс між складністю реалізації та надзвичайно високою продуктивністю на більшості типів тестів, алгоритм Дініца є вибором за замовчуванням для багатьох досвідчених програмістів.

Завдання для самостійної роботи

1. Реалізуйте оптимізацію з вказівниками для DFS-частини вашого алгоритму Дініца. Порівняйте продуктивність з версією без цієї оптимізації.

2. Напишіть програму, яка візуалізує роботу алгоритму Дініца, показуючи на кожній фазі залишкову мережу, потім шарувату мережу, а потім шляхи, що знаходяться DFS.

3. Застосуйте вашу реалізацію алгоритму Дініца для розв’язання задачі про максимальне паросполучення у дводольному графі. Порівняйте швидкість з попередніми методами.

4. Дослідіть, як змінюється кількість фаз алгоритму для різних типів графів (довгі “тонкі” мережі, широкі “приземкуваті” мережі тощо).

5. Модифікуйте алгоритм, щоб він не лише знаходив величину потоку, а й сам мінімальний розріз.

6. Спробуйте реалізувати алгоритм Дініца, використовуючи ітеративний, а не рекурсивний DFS.

7. Дослідіть поведінку алгоритму на мережах, де є ребра з нульовою пропускною здатністю.
8. Напишіть стрес-тест, який генерує складні випадки для алгоритму і перевіряє коректність відповіді, порівнюючи її з результатом роботи повільнішого, але надійнішого алгоритму (напр. Едмондса-Карпа).
9. Адаптуйте алгоритм для знаходження потоку в мережі з кількома джерелами і стоками.
10. Прочитайте про алгоритм “проштовхування-перепозначення” і напишіть короткий есей, що порівнює його ідеологію з ідеологією алгоритму Дініца.

Теми лабораторних робіт

1. Реалізація алгоритму Дініца та порівняльний аналіз його продуктивності з алгоритмом Едмондса-Карпа на різних класах графів.
2. Застосування алгоритму Дініца для вирішення задачі про максимальне паросполучення та порівняння з іншими методами.
3. Створення інтерактивного візуалізатора для покрокової демонстрації роботи алгоритму Дініца (побудова шаруватої мережі, пошук блокуючого потоку).
4. Моделювання логістичних мереж з високою пропускною здатністю за допомогою алгоритму Дініца.
5. Вирішення задачі “бейсбол на вибування” (Baseball Elimination Problem), яка зводиться до задачі про максимальний потік.
6. Розробка програми для аналізу надійності складних інфраструктурних мереж (напр., енергетичних), де швидкість розрахунку критична.
7. Оптимізація реалізації алгоритму Дініца для розріджених графів.
8. Реалізація алгоритму пошуку потоку мінімальної вартості, що використовує алгоритм Дініца як підпрограму.
9. Застосування алгоритмів потоку для аналізу зображень (задача Image Segmentation).

10. Розробка бібліотеки графових алгоритмів, що включає кілька методів пошуку максимального потоку з єдиним інтерфейсом.

Максимальний потік. Метод проштовхування–підняття мітки

До цього моменту всі розглянуті нами алгоритми (Форда-Фалкерсона, Едмондса-Карпа, Дініца) дотримувалися однієї спільної ідеології: вони працювали з коректним потоком і намагалися ітеративно покращити його, знаходячи доповнюючі шляхи від джерела до стоку. Це інтуїтивно зрозумілий підхід, що нагадує поступове наповнення мережі.

Зараз ми познайомимося з кардинально іншим підходом — методом проштовхування–підняття мітки (Push-Relabel). Цей метод відмовляється від ідеї підтримки коректного потоку на проміжних етапах. Натомість він моделює більш "фізичний" процес, схожий на рух води під дією гравітації.

Уявіть, що кожна вершина має певну "висоту", а потік — це вода. Вода завжди тече з вищої точки в нижчу. Метод починається з того, що джерело "затоплює" своїх найближчих сусідів, створюючи в них надлишок потоку. Далі ці "переповнені" вершини намагаються "проштовхнути" свій надлишок до сусідів, що знаходяться нижче. Якщо вершина "застрягла" зі своїм надлишком (бо всі сусіди знаходяться на тій же або вищій висоті), ми "піднімаємо" її, збільшуючи її висоту, щоб вона могла продовжити проштовхувати потік. Цей процес триває, доки вся "зайва" вода не стече в стік або не повернеться назад до джерела.

1. Теоретичні відомості

Цей метод оперує новими поняттями, які відрізняються від тих, що ми використовували в алгоритмах на основі доповнюючих шляхів.

1.1. Передпотік та надлишковий потік

Передпотік — це функція f, схожа на потік, але з однією важливою відмінністю. Вона так само задовольняє умовам обмеження пропускної здатності та антисиметрії, але послаблює умову збереження потоку.

- Умова збереження потоку (послаблена): Для будь-якої вершини u, крім джерела s, сумарний потік, що входить у вершину, не може бути меншим за сумарний потік, що виходить з неї.

$$\forall u \in V \backslash \{s\}: \quad \sum_{v \in V} f(v, u) \geq 0$$

Це означає, що проміжні вершини можуть “накопичувати” потік.

Надлишковий потік (Excess Flow) для вершини u — це величина, на яку вхідний потік перевищує вихідний. Позначається як $e(u)$.

$$e(u) = \sum_{v \in V} f\,(v, u)$$

– Якщо $e(u) > 0$, вершина u називається переповненою або активною.

– Мета алгоритму — позбутися всіх надлишкових потоків у проміжних вершинах, перетворивши передпотік на коректний максимальний потік.

1.2. Функція висоти (мітка)

Функція висоти $h: V \to \mathbb{N}$ — це “мітка”, що присвоюється кожній вершині. Вона керує напрямком, у якому можна проштовхувати потік.

– Потік можна проштовхнути від вершини u до вершини v лише якщо u знаходиться рівно на один рівень вище за v, тобто $h(u) = h(v) + 1$.

– Висота джерела $h(s)$ та стоку $h(t)$ є фіксованими:

  - $h(s) = |V|$ (кількість вершин у графі).
  - $h(t) = 0$.

– Висоти всіх інших вершин на початку дорівнюють 0.

Висота $h(u)$ є оцінкою відстані від u до стоку t. Оскільки найдовший простий шлях не може перевищувати $|V| - 1$, встановивши $h(s) = |V|$, ми гарантуємо, що потік ніколи не зможе повернутися до джерела по шляху з “правильних” ребер (де кожне наступне на 1 нижче).

1.3. Основні операції

Алгоритм базується на двох простих локальних операціях, які застосовуються до активних вершин.

11. Проштовхування Операція `Push(u, v)` переміщує потік з активної вершини u до її сусіда v.

– Умови застосування:

  - Вершина u є активною ($e(u) > 0$).

- Залишкова пропускна здатність ребра (u, v) є додатною $(c_f(u, v) = c(u, v) - f(u, v) > 0)$.
- Вершина u на один рівень вища за v $(h(u) = h(v) + 1)$.

– Дія: Проштовхнути потік об'ємом $\Delta = \min(e(u), c_f(u, v))$.

- $f(u, v) \leftarrow f(u, v) + \Delta$
- $f(v, u) \leftarrow f(v, u) - \Delta$
- $e(u) \leftarrow e(u) - \Delta$
- $e(v) \leftarrow e(v) + \Delta$

5. Підняття мітки Операція `Relabel(u)` збільшує висоту активної вершини u, яка "застрягла" і не може проштовхнути свій надлишок.

– Умови застосування:

- Вершина u є активною $(e(u) > 0)$.
- Для всіх сусідів v, до яких є ребро в залишковій мережі $(c_f(u, v) > 0)$, виконується умова $h(u) \leq h(v)$.

– Дія: Збільшити висоту u до мінімально можливого рівня, який дозволить їй у майбутньому проштовхнути потік.

- $h(u) \leftarrow 1 + \min h(v) \mid c_f(u, v) > 0$

1.4. Загальний алгоритм

Псевдокод ініціалізації

```
function Initialize-Preflow(G, s):
  // Ініціалізуємо висоти та надлишки
  for each vertex v in G.V:
    h[v] = 0
    e[v] = 0
  // Ініціалізуємо потік
  for each edge (u, v) in G.E:
    f(u, v) = 0
    f(v, u) = 0

  // Встановлюємо висоту джерела
  h[s] = |V|
```

```
  // Насичуємо ребра, що виходять з джерела s
  for each neighbor v of s:
    // Проштовхуємо максимальний можливий потік
    flow_to_push = c(s, v)
    f(s, v) = flow_to_push
    f(v, s) = -flow_to_push
    // Створюємо надлишок у сусідів
    e[v] = e[v] + flow_to_push
    // Зменшуємо "надлишок" у джерела (він стане від'ємним)
    e[s] = e[s] - flow_to_push
```

Псевдокод загального методу

```
function Generic-Push-Relabel(G, s, t):
  Initialize-Preflow(G, s)

  // Поки існують активні вершини (крім s і t)
  while there exists an active vertex u in V \ {s, t}:
    // Вибираємо довільну активну вершину u

    // Шукаємо, куди можна проштовхнути потік
    found_push_target = false
    for each neighbor v of u:
      // Перевіряємо умови для Push
      if c(u, v) - f(u, v) > 0 and h[u] == h[v] + 1:
        Push(u, v)
        found_push_target = true
        break // Виконали одне проштовхування і шукаємо нову активну
вершину

    // Якщо не вдалося нікуди проштовхнути потік
    if not found_push_target:
      Relabel(u)

  // Коли активних вершин немає, потік в стоці t є максимальним
  return e[t]
```

1.5. Аналіз складності та евристики

- Кількість операцій **Relabel**: Висота будь-якої вершини не може перевищувати $2|V| - 1$. Оскільки висота лише зростає, кожна з $|V| - 2$ проміжних вершин може бути піднята $O(V)$ разів. Загальна кількість операцій **Relabel** — $O(V^2)$.

- Кількість операцій **Push**: Аналіз складніший. Можна виділити два типи проштовхувань:

- Насичуючі (Saturating): Коли $\Delta = c_f(u, v)$. Таких проштовхувань не більше $O(V \cdot E)$.
- Ненасичуючі (Non-saturating): Коли $\Delta = e(u)$. Таких може бути багато.

– Загальна складність загального методу може сягати $O(V^2 \cdot E)$.

Проте продуктивність сильно залежить від порядку, в якому ми вибираємо активні вершини. Декілька популярних евристик значно покращують складність:

– FIFO (First-In, First-Out): Активні вершини обробляються в порядку їх появи (за допомогою черги). Складність — $O(V^3)$.

– Highest-Label Selection: Завжди вибирається активна вершина з найбільшою висотою. Складність — $O(V^2\sqrt{E})$.

– Discharge: Операція, що об'єднує всі можливі `Push` та одне `Relabel` для однієї вершини, доки її надлишок не стане нулем. Це спрощує реалізацію.

2. Область застосування

Метод проштовхування-підняття мітки є одним з найефективніших.

– Теоретична інформатика: Алгоритми цього сімейства мають одні з найкращих асимптотичних оцінок складності.

– Паралельні обчислення: Локальний характер операцій `Push` і `Relabel` (вони залежать лише від вершини та її найближчих сусідів) робить цей метод дуже придатним для розпаралелювання.

– Високопродуктивні реалізації: У багатьох бібліотеках графових алгоритмів для пошуку максимального потоку використовуються саме просунуті версії методу Push-Relabel.

– Комп'ютерний зір: Як і інші потокові алгоритми, він активно використовується в задачах сегментації зображень.

3. Приклад застосування

Візьмемо нашу стандартну мережу: c(S,A)=10, c(S,B)=10, c(A,B)=2, c(A,T)=10, c(B,T)=10. $|V| = 4$.

Крок 0: Ініціалізація

- `h` = {S:4, A:0, B:0, T:0}
- `e` = {S:0, A:0, B:0, T:0}
- `f` скрізь 0.
- Проштовхуємо потік з S:
  - `Push(S, A)` на 10. `f(S,A)=10, e(A)=10, e(S)=-10`.
  - `Push(S, B)` на 10. `f(S,B)=10, e(B)=10, e(S)=-20`.
- Початковий стан: `e`={S:-20, A:10, B:10, T:0}, `h`={S:4, A:0, B:0, T:0}.
- Активні вершини: {A, B}.

Ітерація 1

- Вибираємо активну вершину A ($e(A) = 10$).
- Дивимося на сусідів A в залишковій мережі: T ($c_f(A,T) = 10$) і B ($c_f(A,B) = 2$).
- `h(A)=0, h(T)=0, h(B)=0`. Умова `h(A) = h(v)+1` не виконується ні для кого.
- Дія: `Relabel(A)`. Мінімальна висота сусідів (T і B) дорівнює 0. Нова висота `h(A) = 1 + 0 = 1`.
- Стан: `e`={S:-20, A:10, B:10, T:0}, `h`={S:4, A:1, B:0, T:0}. Активні: {A, B}.

Ітерація 2

- Вибираємо активну вершину B ($e(B) = 10$).
- Сусіди в залишковій мережі: T ($c_f(B,T) = 10$).
- `h(B)=0, h(T)=0`. Умова `h(B) = h(T)+1` не виконується.
- Дія: `Relabel(B)`. Мін. висота сусіда (T) дорівнює 0. Нова висота `h(B) = 1 + 0 = 1`.
- Стан: `e`={S:-20, A:10, B:10, T:0}, `h`={S:4, A:1, B:1, T:0}. Активні: {A, B}.

Ітерація 3

- Вибираємо активну вершину A ($e(A) = 10$).

- Сусід T: $\mathtt{h(A)=1,\ h(T)=0}$. Умова $\mathtt{h(A)\ =\ h(T)+1}$ виконується!
- Дія: $\mathtt{Push(A,\ T)}$. Об'єм $\Delta = \min(e(A), c_f(A,T)) = \min(10,10) = 10$.
- $\mathtt{e(A)\ \backslash leftarrow\ 10-10=0}$. $\mathtt{e(T)\ \backslash leftarrow\ 0+10=10}$.
- Стан: $\mathtt{e}$={S:-20, A:0, B:10, T:10}, $\mathtt{h}$={S:4, A:1, B:1, T:0}. Активна: {B}.

Ітерація 4

- Вибираємо активну вершину B ($e(B) = 10$).
- Сусід T: $\mathtt{h(B)=1,\ h(T)=0}$. Умова $\mathtt{h(B)\ =\ h(T)+1}$ виконується!
- Дія: $\mathtt{Push(B,\ T)}$. Об'єм $\Delta = \min(e(B), c_f(B,T)) = \min(10,10) = 10$.
- $\mathtt{e(B)\ \backslash leftarrow\ 10-10=0}$. $\mathtt{e(T)\ \backslash leftarrow\ 10+10=20}$.
- Стан: $\mathtt{e}$={S:-20, A:0, B:0, T:20}, $\mathtt{h}$={S:4, A:1, B:1, T:0}.

Завершення

- Активних вершин (крім s і t) більше немає. Алгоритм завершено.
- Результат — надлишковий потік у стоці $\mathtt{e(T)=20}$. Це і є величина максимального потоку.

4. Переваги та недоліки

Переваги

- Висока теоретична ефективність: Алгоритми цього сімейства (з евристиками) мають одні з найкращих доведених асимптотичних оцінок складності, наприклад, $O(V^3)$ або $O(V^2\sqrt{E})$.
- Відсутність пошуку шляхів: Метод не потребує пошуку шляхів від s до t, що робить його структуру принципово іншою.
- Придатність до розпаралелювання: Локальний характер операцій дозволяє легко розпаралелювати обчислення, обробляючи кілька активних вершин одночасно.

Недоліки

- Висока складність для розуміння та реалізації: Концепції передпотоку, висот та їхня взаємодія є менш інтуїтивними, ніж ідея доповнюючих шляхів.

– Складність налагодження: Через велику кількість локальних змін стану (висоти, надлишки) відстежити помилку в реалізації може бути дуже складно.

– На практиці може бути повільнішим: Незважаючи на кращу асимптотику, на деяких класах графів простіший алгоритм Дініца може працювати швидше через менші константні фактори.

Контрольні запитання

– У чому головна ідеологічна відмінність методу проштовхування–підняття мітки від методів, що базуються на доповнюючих шляхах?

– Що таке “передпотік” і чим він відрізняється від коректного потоку?

– Яку роль виконує “функція висоти” (мітка)?

– Назвіть точні умови, за яких можна виконати операцію `Push(u, v)`.

– Назвіть точні умови, за яких потрібно виконати операцію `Relabel(u)`.

– Чому висота джерела $h(s)$ встановлюється рівною $|V|$?

– Як евристики вибору активної вершини впливають на складність алгоритму?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація методу проштовхування–підняття мітки для пошуку максимального потоку.

2. Мета

– Опанувати принципово інший підхід до розв’язання задачі про максимальний потік.

– Набути навичок реалізації алгоритмів, що працюють на основі локальних правил і змін стану.

– Зрозуміти концепції передпотоку, надлишкового потоку та функції висоти.

3. Завдання

– Реалізувати структури даних для представлення потокової мережі, а також для зберігання надлишків `e` та висот `h` для кожної вершини.

– Реалізувати функцію ініціалізації `Initialize-Preflow`.

– Реалізувати дві основні операції: `Push(u, v)` та `Relabel(u)`.

– Реалізувати головний цикл алгоритму, який керує вибором активних вершин та застосуванням операцій.

– Програма повинна читати опис мережі з файлу та виводити величину максимального потоку.

4. Опис

Ця лабораторна робота є найскладнішою в циклі робіт про максимальний потік. Вона вимагає від студента не лише написання коду, а й глибокого розуміння нетривіальної логіки алгоритму. Основний акцент — на коректному управлінні станом (надлишками та висотами) та правильному застосуванні двох базових операцій.

5. Порядок виконання

– Структури даних: Вам знадобляться масиви або словники для `h[V]`, `e[V]`, `capacity[V][V]` та `flow[V][V]`.

– Ініціалізація: Напишіть функцію, яка встановлює початкові висоти та “затоплює” сусідів джерела, створюючи перші активні вершини.

– Реалізація операцій: Створіть окремі, добре тестовані функції для `Push` та `Relabel`. Переконайтеся, що вони коректно оновлюють всі відповідні значення (потоки, надлишки, висоти).

– Головний цикл: Реалізуйте цикл `while`, що працює, поки є активні вершини. Всередині циклу організуйте вибір активної вершини та застосування до неї операцій `Push` або `Relabel`.

– Вибір активних вершин: Для спрощення можна просто ітерувати по всіх вершинах і шукати першу активну. Для більш просунутої реалізації можна використати список або чергу активних вершин.

– Тестування: Ретельно тестуйте програму на невеликих прикладах, де можна вручну простежити зміни висот і надлишків. Порівняйте результати з попередніми алгоритмами.

6. Результат

Результатом роботи є програма, що реалізує метод проштовхування–підняття мітки, та звіт, що пояснює логіку роботи, вибір структур даних та результати тестування.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
  - Функція ініціалізації передпотоку.
  - Функції `Push` та `Relabel`.
  - Основний цикл, що керує процесом.
- Продемонструвати роботу додатку:
  - На кількох тестових прикладах.
  - Пояснити покроково зміни висот `h` та надлишків `e` для невеликого прикладу на 2-3 ітераціях.
- Бути готовим відповісти на питання:
  - Питання з розділу “Контрольні запитання”.
  - Яку стратегію вибору активної вершини ви реалізували?
  - Як ваша програма підтримує залишкову мережу для перевірки умов операцій?
  - Які, на вашу думку, основні складнощі при налагодженні цього алгоритму?

Тести

1. Метод проштовхування–підняття мітки працює з поняттям:

a) Доповнюючого шляху

b) Кістякового дерева

c) Передпотоку

d) Гамільтонового циклу

2. Операція `Push(u, v)` можлива, лише якщо, крім іншого, виконується умова:

a) `h[u] > h[v]`

b) `h[u] < h[v]`

c) `h[u] == h[v] + 1`

d) `h[u] == h[v]`

3. Операція `Relabel(u)` застосовується, коли:

a) Вершина `u` досягла стоку.

b) Вершина `u` активна, але не може проштовхнути потік до жодного сусіда.

c) Надлишковий потік вершини `u` дорівнює нулю.

d) Висота вершини `u` є максимальною.

4. На початку роботи алгоритму висота джерела `h(s)` встановлюється рівною:

a) 0

b) 1

c) |E| (кількість ребер)

d) |V| (кількість вершин)

5. Алгоритм завершує роботу, коли:

a) Всі вершини мають однакову висоту.

b) Немає жодної активної вершини (крім, можливо, стоку).

c) Потік через кожне ребро дорівнює його пропускній здатності.

d) Виконано $|V|$ операцій `Relabel`.

Тести підвищеної складності

1. Чому висота будь-якої активної вершини ніколи не може перевищити $2|V| - 1$?

a) Тому що найдовший шлях до джерела, куди може повернутися потік, має довжину $|V| - 1$, а висота джерела $|V|$.

b) Це емпіричне обмеження для прискорення роботи.

c) Це максимальне значення, яке може зберігати стандартний тип даних.

d) Це пов’язано з кількістю ребер у графі.

2. Яка евристика вибору активної вершини призводить до однієї з найкращих теоретичних оцінок складності $O(V^2\sqrt{E})$?

a) Вибір випадкової активної вершини.

b) FIFO (перший прийшов — перший вийшов).

c) LIFO (останній прийшов — перший вийшов).

d) Вибір активної вершини з найбільшою висотою.

3. “Насичуюче” проштовхування (saturating push) — це операція `Push(u, v)`, де обсяг проштовхнутого потоку дорівнює:

a. `e(u)` (надлишку вершини u)

b. `c_f(u, v)` (залишковій пропускній здатності ребра)

c. `e(u) + c_f(u, v)`

d. `h(u) - h(v)`

4. У порівнянні з алгоритмом Дініца, метод проштовхування-підняття мітки:

a) Завжди простіший в реалізації.

b) Має кращі перспективи для розпаралелювання.

c) Завжди працює швидше на практиці.

d) Базується на тій самій ідеї доповнюючих шляхів.

5. Після завершення роботи алгоритму надлишковий потік `e(t)` у стоці — це:

a) Величина максимального потоку.

b) Помилка, надлишок має бути нульовим.

c) Кількість ітерацій алгоритму.

d) Середня висота всіх вершин.

6. Що станеться, якщо на початку встановити `h(s) = |V|-1` замість `h(s) = |V|`?

a) Нічого не зміниться.

b) Алгоритм може працювати некоректно, оскільки потік потенційно може “повернутися” до джерела.

c) Алгоритм працюватиме швидше.

d) Алгоритм не зможе виконати ініціалізацію.

7. “Операція `Discharge(u)`” — це:

a) Інша назва для операції `Push`.

b) Процедура, що повторює `Push` та `Relabel` для вершини `u`, доки її надлишок не стане нульовим.

c) Видалення вершини `u` з графа.

d) Встановлення `e(u)` в нуль примусово.

8. Яка основна перевага методу Push-Relabel у контексті динамічних графів (де ребра можуть додаватися/видалятися)?

a) Жодних переваг немає.

b) Його можна легко адаптувати, оскільки зміни є локальними і не вимагають повного перерахунку шляхів.

c) Він може передбачати майбутні зміни.

d) Він не працює на динамічних графах.

9. Якщо в ході виконання алгоритму `h(u) > |V|`, що це означає?

a) Вершина `u` більше не може проштовхнути потік до стоку t.

b) Це нормальна ситуація для деяких вершин.

c) В реалізації допущена помилка.

d) Мережа не має потоку.

10. У фінальному стані (коли знайдено максимальний потік), які з тверджень про висоти є правильними?

a) Всі висоти дорівнюють нулю.

b) Висоти утворюють точну відстань до стоку в залишковій мережі.

c) `h(s)` залишається $|V|$, `h(t)` залишається 0.

d) `h(u) = h(v)` для всіх сусідніх вершин.

Завдання для самостійної роботи

1. Реалізуйте евристику FIFO для вибору активних вершин. Вам знадобиться черга для зберігання активних вершин.
2. Реалізуйте операцію **`Discharge`** і перебудуйте головний цикл алгоритму на її основі.
3. Створіть візуалізатор, який показує граф, а для кожної вершини — її поточну висоту та величину надлишкового потоку. Анімація операцій **`Push`** та **`Relabel`**.
4. Дослідіть, як змінюється максимальна висота, якої досягають вершини, залежно від структури графа.
5. Порівняйте реальний час роботи вашої реалізації Push-Relabel з реалізацією алгоритму Дініца на однакових наборах тестів.
6. Напишіть функцію, яка після завершення алгоритму знаходить мінімальний розріз, використовуючи фінальні висоти вершин.
7. Адаптуйте алгоритм для мереж, де вершини також мають пропускну здатність.
8. Прочитайте про “Gap Heuristic” для методу Push-Relabel і спробуйте реалізувати її.
9. Реалізуйте евристику “Highest-Label Selection”. Вам знадобиться структура даних, що дозволяє ефективно знаходити активну вершину з максимальною висотою.

10. Напишіть програму, яка перевіряє, чи є задана пара функцій `(h, e)` коректними мітками висоти та надлишкового потоку для заданого передпотоку `f`.

Теми лабораторних робіт

1. Реалізація методу проштовхування-підняття мітки з евристикою FIFO та порівняння його продуктивності з алгоритмом Дініца.
2. Розробка паралельної версії методу проштовхування–підняття мітки з використанням багатопотоковості.
3. Застосування методу проштовхування-підняття мітки для вирішення задачі сегментації зображень.
4. Створення інтерактивного навчального посібника-візуалізатора для методу проштовхування-підняття мітки.
5. Порівняльний аналіз різних евристик вибору активних вершин (FIFO, Highest-Label, LIFO) для методу проштовхування-підняття мітки.
6. Реалізація алгоритму для динамічних графів, що ефективно оновлює максимальний потік після додавання/видалення ребра.
7. Застосування методу для вирішення задачі про потік мінімальної вартості.
8. Розробка бібліотеки для пошуку максимального потоку, що включає алгоритми Едмондса-Карпа, Дініца та проштовхування-підняття мітки з можливістю вибору.
9. Аналіз продуктивності проштовхування-підняття мітки на реальних мережевих даних (напр., дорожні мережі, інтернет-топології).
10. Реалізація однієї з просунутих версій алгоритму, наприклад, з використанням динамічних дерев для покращення асимптотичної складності.

Відповіді: 1. - c), 2. - c), 3. - b), 4. - d),5.- b).

Відповіді на тести підвищеної складності

1. a) Висота $h(u)$ є нижньою межею відстані до s або t у залишковій мережі. Якщо потік може повернутись до s, то $h(u) \leq h(s) + \text{dist}_f(u, s) = |V| + (|V| - 1) = 2|V| - 1$.

2. d) Евристика вибору активної вершини з найбільшою висотою є однією з ключових для досягнення кращих оцінок складності.

3. b) Насичуючим називається проштовхування, яке повністю використовує залишкову пропускну здатність ребра. Воно обмежене саме ребром, а не надлишком.

4. b) Хоча його складніше реалізувати, його локальна природа робить його значно придатнішим до розпаралелювання, ніж алгоритми, що шукають глобальні шляхи.

5. a) Після завершення роботи алгоритму надлишковий потік у всіх проміжних вершинах дорівнює нулю. Весь надлишок, що не повернувся до джерела, накопичується в стоці, і це значення є величиною максимального потоку.

6. b) Висота `h(s)=|V|` гарантує, що не існує шляху з s до s у залишковій мережі, де кожна наступна вершина має меншу висоту. Якщо `h(s)=|V|-1`, такий шлях теоретично можливий.

7. b) Це поширена техніка реалізації, яка інкапсулює логіку обробки однієї активної вершини в одній процедурі.

8. b) Зміна ребра впливає лише на стан сусідніх вершин. Можна локально скоригувати потік, висоти та надлишки, не запускаючи весь алгоритм з нуля, що набагато ефективніше, ніж у методах на основі шляхів.

9. a) Висота $h(u)$ може бути інтерпретована як відстань. Якщо вона більша за $|V|$, це означає, що вершина “відрізана” від стоку, оскільки будь-який шлях до стоку (де $h(t) = 0$) був би довшим за найдовший можливий простий шлях.

10. c) Алгоритм змінює висоти, але початкові значення `h(s)` та `h(t)` є інваріантами. Фінальні висоти інших вершин дають інформацію про мінімальний розріз.

Паросполучення

Після вивчення способів з'єднання вершин (кістякові дерева) та транспортування потоків (максимальний потік), ми переходимо до задачі оптимального призначення або паросполучення.

Уявіть, що вам потрібно сформувати пари: призначити працівників на відповідні їм вакансії, студентів до курсових проєктів, чи пілотів до літаків. У кожному з цих випадків ви хочете створити якомога більше пар, дотримуючись певних обмежень (наприклад, один працівник може виконувати лише одну роботу одночасно). У теорії графів ця задача моделюється як пошук максимального паросполучення — найбільшого набору ребер, в якому жодні два ребра не мають спільної вершини.

Цей розділ ми розділимо на дві великі частини, що відповідають двом класам графів, для яких підходи до розв'язання кардинально відрізняються:

1. Паросполучення в дводольних графах. Це особливий, але дуже поширений випадок, де вершини можна розділити на дві незалежні долі (наприклад, "працівники" та "вакансії"). Ми розглянемо зв'язок цієї задачі з уже вивченою задачею про максимальний потік, а потім заглибимося в алгоритм Гопкрофта-Карпа — надзвичайно ефективний метод, який довгий час був найшвидшим для розв'язання цієї проблеми.

2. Паросполучення в загальних графах. Що, якщо граф не є дводольним? Наявність циклів непарної довжини руйнує прості підходи, що працювали для дводольних графів. Тут на сцену виходить один з найелегантніших алгоритмів у теорії графів — алгоритм "квітки". Ми детально розберемо його ідею: знаходити цикли непарної довжини ("квітки"), "стискати" їх в одну вершину і рекурсивно розв'язувати задачу, що відкрило шлях до ефективного пошуку паросполучень у будь-яких графах.

Вивчення цих алгоритмів не лише познайомить вас із класичними методами оптимізації, але й продемонструє, як структура графа визначає складність задачі та витонченість методів її розв'язання.

Паросполучення в дводольних графах. Алгоритм Гопкрофта-Карпа

У попередніх розділах ми дослідили потужні інструменти для роботи з потоками, зазначивши, що задачу пошуку максимального паросполучення у дводольному графі можна звести до задачі про максимальний потік. Це надійний підхід, але він є дещо надлишковим. Задача про паросполучення має специфічну структуру, яку можна використати для створення значно швидших, спеціалізованих алгоритмів.

Алгоритм Гопкрофта-Карпа є вершиною ефективності для цієї задачі. Він залишався асимптотично найшвидшим методом для пошуку максимального паросполучення в незважених дводольних графах протягом десятиліть. Його ідея разюче нагадує алгоритм Дініца, але адаптована до специфіки паросполучень: замість того, щоб знаходити один доповнюючий шлях за раз, він знаходить цілий набір найкоротших доповнюючих шляхів за одну фазу.

1. Теоретичні відомості

Перш ніж перейти до алгоритму, давайте формалізуємо основні поняття, специфічні для паросполучень.

1.1. Базові визначення

- Дводольний граф: Це граф $G = (V, E)$, множину вершин якого V можна розділити на дві неперетинні підмножини U і W ($V = U \cup W, U \cap W = \emptyset$) так, що кожне ребро з E з'єднує вершину з U з вершиною з W.
- Паросполучення: Це підмножина ребер $M \subseteq E$, в якій жодні два ребра не мають спільної вершини.
- Максимальне паросполучення: Це паросполучення з найбільшою можливою кількістю ребер.
- Вільна вершина: Вершина, яка не є кінцем жодного ребра з поточного паросполучення M.
- Чергувальний шлях: Простий шлях у графі, ребра якого по черзі належать і не належать до паросполучення M.
- Доповнюючий шлях: Це чергувальний шлях, початкова і кінцева вершини якого є вільними.

Теорема Берже: Паросполучення M є максимальним тоді і тільки тоді, коли в графі не існує доповнюючого шляху відносно M.

Ця теорема є фундаментальною. Вона означає, що ми можемо збільшувати розмір нашого паросполучення, доки можемо знаходити доповнюючі шляхи. Якщо ми "перевернемо" ребра вздовж доповнюючого шляху (тобто ребра, що не були в M, додамо до M, а ті, що були, — видалимо), розмір паросполучення збільшиться рівно на одиницю.

Простий алгоритм міг би на кожному кроці шукати один доповнюючий шлях (за допомогою DFS або BFS) і перевертати його. Це працює, але може бути повільно. Алгоритм Гопкрофта-Карпа робить це набагато ефективніше.

1.2. Ідея алгоритму Гопкрофта-Карпа

Алгоритм працює у Крок х. У кожній фазі він знаходить максимальний набір вершинно-неперетинних та найкоротших доповнюючих шляхів.

1. Пошук довжини: За допомогою BFS визначається довжина k найкоротшого доповнюючого шляху.

2. Пошук набору шляхів: За допомогою DFS шукаються всі можливі доповнюючі шляхи довжиною рівно k, з умовою, що жодна вершина не може належати до двох різних шляхів у цьому наборі.

3. Збільшення паросполучення: Всі знайдені шляхи одночасно "перевертаються", збільшуючи розмір паросполучення на кількість знайдених шляхів.

4. Фази повторюються. Ключова властивість полягає в тому, що довжина найкоротших доповнюючих шляхів строго зростає з кожною фазою.

1.3. Реалізація алгоритму

Крок 1: Побудова шарів за допомогою BFS

Ми запускаємо BFS одночасно від усіх вільних вершин з однієї долі, скажімо U.

– Мета BFS — обчислити відстань (кількість ребер в чергувальному шляху) від вільної вершини з U до кожної іншої вершини. Ми будемо зберігати ці відстані в масиві `dist`.

– BFS рухається по черзі:

- З вершини $u \in U$ по ребру, що не належить паросполученню M.
- З вершини $w \in W$ по ребру, що належить паросполученню M.

– BFS зупиняється, як тільки досягає перших вільних вершин у долі W. Довжина шляху до них і буде довжиною k найкоротших доповнюючих шляхів у цій фазі. Якщо жодна вільна вершина в W не досяжна, доповнюючих шляхів більше немає, і алгоритм завершується.

Псевдокод побудови шарів (BFS)

```
function BFS_BuildLayers(G, U, W, matching, dist):
  // dist - масив відстаней
  // matching[v] = u означає, що ребро (u,v) в паросполученні
  queue = new Queue()

  // Ініціалізуємо відстані
  for each u in U:
    if u is free (not in matching):
      dist[u] = 0
      queue.enqueue(u)
    else:
      dist[u] = ∞

  dist[NIL] = ∞ // NIL - спеціальний маркер для вільних вершин W

  while not queue.is_empty():
    u = queue.dequeue()
    if dist[u] < dist[NIL]:
      // Рухаємось по ребрах, що не в паросполученні
      for each neighbor w of u:
        // Якщо w спарована і шлях до її пари ще не знайдено
        if dist[matching[w]] == ∞:
          dist[matching[w]] = dist[u] + 1
          queue.enqueue(matching[w])

  // Повертаємо true, якщо знайдено шлях до хоча б однієї вільної вер
шини з W
  return dist[NIL] != ∞
```

Крок 2: Пошук набору шляхів за допомогою DFS

Після того, як ми обчислили відстані (побудували “шари”), ми запускаємо DFS з кожної вільної вершини в U.

– DFS рухається від вершини $u \in U$ до сусідньої вершини $w \in W$ лише якщо $\text{dist}[matching[w]] = \text{dist}[u] + 1$. Це гарантує, що ми рухаємося по найкоротшому шляху.

– Якщо DFS досягає вільної вершини в W, це означає, що знайдено доповнюючий шлях. Ми “перевертаємо” його (оновлюємо паросполучення) і повертаємо **`true`**.

– Кожна вершина використовується не більше одного разу в межах однієї фази DFS.

Псевдокод пошуку шляхів (DFS)

```
function DFS_FindPaths(u, matching, dist, G):
  if u != NIL:
    // Рухаємось по шарах
    for each neighbor w of u:
      // Перевіряємо, чи веде ребро до наступного шару
      if dist[matching[w]] == dist[u] + 1:
        // Рекурсивно шукаємо продовження шляху
        if DFS_FindPaths(matching[w], matching, dist, G):
          // Шлях знайдено! Оновлюємо паросполучення.
          matching[w] = u
          matching[u] = w
          return true

    // Якщо з вершини u не вдалося знайти шлях
    dist[u] = ∞ // Позначаємо, що ця вершина "тупикова"
    return false

  return true
```

Псевдокод основного алгоритму

```
function HopcroftKarp(G, U, W):
  // Ініціалізуємо порожнє паросполучення і відстані
  matching = {} // Порожній словник/масив
  dist = {}
  matching_size = 0

  // Поки BFS знаходить доповнюючі шляхи
  while BFS_BuildLayers(G, U, W, matching, dist):
    // Для кожної вільної вершини в U намагаємось знайти шлях
    for each u in U:
      if u is free:
        if DFS_FindPaths(u, matching, dist, G):
```

```
            matching_size = matching_size + 1

    return matching_size, matching
```

1.4. Аналіз складності

- Кількість фаз: Це найскладніша частина доведення. Можна показати, що довжина найкоротшого доповнюючого шляху строго зростає з кожною фазою. Завдяки цьому, загальна кількість фаз не перевищує $O(\sqrt{|V|})$.
- Складність однієї фази: Кожна Крок складається з одного запуску BFS та одного “великого” DFS. Обидві операції відвідують кожне ребро та вершину обмежену кількість разів, тому складність однієї фази — $O(|E|)$.
- Загальна складність:

$$\text{(Кількість фаз)} \times \text{(Складність фази)} = O(\sqrt{V}) \times O(E) = O(E\sqrt{V})$$

Це значно краще за $O(V \cdot E)$ (складність зведення до потоку і розв’язання алгоритмом Дініца для загального випадку, хоча для одиничних мереж Дініц теж дає $O(E\sqrt{V})$) та $O(V \cdot E^2)$ для Едмондса-Карпа.

2. Область застосування

Алгоритм Гопкрофта-Карпа є найкращим вибором для пошуку максимального паросполучення саме в незважених дводольних графах.

- Задачі про призначення: Класичне застосування. Призначення працівників на задачі, студентів на проекти, водіїв на маршрути, якщо кожен зв’язок є бінарним (підходить/не підходить).
- Планування та розклади: Створення розкладу, де потрібно зіставити часові слоти з подіями або викладачами.
- Обчислювальна біологія: Аналіз взаємодій між білками або генами, які можна змоделювати як дводольний граф.
- Основа для інших алгоритмів: Використовується як підпрограма в більш складних алгоритмах на графах.
- Конкурентне програмування: Є стандартним розв’язком для відповідних задач завдяки своїй високій ефективності.

3. Приклад застосування

Розглянемо дводольний граф $G = (U \cup W, E)$

- $U = u_1, u_2, u_3, u_4$
- $W = w_1, w_2, w_3, w_4$
- Ребра: $(u_1, w_1), (u_1, w_2), (u_2, w_1), (u_3, w_2), (u_3, w_3), (u_4, w_3)$.

Початковий стан: Паросполучення M порожнє. Всі вершини вільні.

Крок 1

- BFS: Запускаємо BFS з усіх вільних вершин U: u_1, u_2, u_3, u_4.
  - dist(u1, u2, u3, u4) = 0.
  - Шукаємо сусідів в W. Наприклад, з u_1 можна потрапити до w_1 і w_2. Оскільки w_1, w_2 вільні, ми знайшли доповнюючі шляхи довжиною 1. dist(NIL) = 1.
- DFS:
  - Запускаємо DFS з u_1. Він знаходить шлях $u_1 \to w_1$. Додаємо (u_1, w_1) до M. `matching_size=1`.
  - Запускаємо DFS з u_2. Вершина w_1 вже зайнята. Шляху до вільної вершини з u_2 немає.
  - Запускаємо DFS з u_3. Він знаходить шлях $u_3 \to w_2$. Додаємо (u_3, w_2) до M. `matching_size=2`.
  - Запускаємо DFS з u_4. Він знаходить шлях $u_4 \to w_3$. Додаємо (u_4, w_3) до M. `matching_size=3`.
- Кінець Фази 1: $M = (u_1, w_1), (u_3, w_2), (u_4, w_3)$. Вільні вершини: $u_2 \in U$, $w_4 \in W$.

Крок 2

- BFS: Запускаємо BFS з єдиної вільної вершини в U — u_2.
  - `dist(u2) = 0`.
  - З u_2 є ребро до w_1. w_1 не вільна, вона в парі з u_1. Рухаємося по ребру з паросполучення (w_1, u_1). `dist(u1) = dist(u2) + 1 = 1`.

- З u_1 є ребро до w_2. w_2 не вільна, вона в парі з u_3. Рухаємося по ребру з паросполучення (w_2, u_3). `dist(u3) = dist(u1) + 1 = 2`.
- З u_3 є ребро до w_3. w_3 не вільна…
- В результаті BFS не може знайти шлях до жодної вільної вершини в W (єдина вільна там — w_4). BFS повертає `false`.

Завершення:

– Алгоритм завершується, оскільки BFS не знайшов нових доповнюючих шляхів.

– Максимальне паросполучення має розмір 3. $M = (u_1, w_1), (u_3, w_2), (u_4, w_3)$.

4. Переваги та недоліки

Переваги

– Висока ефективність: Складність $O(E\sqrt{V})$ є найкращою відомою для цієї задачі і значно перевершує простіші підходи.

– Теоретична значущість: Алгоритм є важливим теоретичним результатом, що демонструє силу ідеї пошуку кількох шляхів одночасно.

Недоліки

– Складність реалізації: Алгоритм є значно складнішим для коректного впровадження, ніж простий пошук доповнюючого шляху за допомогою одного DFS/BFS.

– Вузька спеціалізація: Він працює лише для незважених дводольних графів. Для зважених або недводольних графів потрібні інші, ще складніші алгоритми (наприклад, угорський алгоритм або алгоритм Блоссом).

Контрольні запитання

– Що таке “доповнюючий шлях” і чому він важливий для пошуку максимального паросполучення?

- У чому полягає ключова перевага алгоритму Гопкрофта-Карпа над простішим алгоритмом, що знаходить один доповнюючий шлях за раз?
- Яку роль виконує BFS в алгоритмі?
- Яку роль виконує DFS в алгоритмі?
- Чим пошук шляхів у Гопкрофта-Карпа схожий на пошук блокуючого потоку в алгоритмі Дініца?
- Чому довжина найкоротших доповнюючих шляхів зростає з кожною фазою?
- Для яких типів графів цей алгоритм не можна застосовувати?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Гопкрофта-Карпа для пошуку максимального паросполучення в дводольному графі.

2. Мета

- Опанувати ефективний алгоритм для фундаментальної задачі про паросполучення.
- Набути практичних навичок у реалізації комбінованих алгоритмів (BFS+DFS).
- Зрозуміти концепцію пошуку максимального набору вершинно-неперетинних шляхів.

3. Завдання

- Реалізувати структуру даних для представлення дводольного графа.
- Реалізувати функцію BFS для побудови шаруватої структури шляхів.
- Реалізувати функцію DFS для пошуку доповнюючих шляхів у шаруватій структурі.
- Об’єднати ці функції в головному циклі алгоритму Гопкрофта-Карпа.
- Програма повинна читати опис графа з файлу та виводити розмір максимального паросполучення і самі ребра, що до нього увійшли.

4. Опис

Ця лабораторна робота вимагає реалізації одного з найскладніших, але й найефективніших класичних графових алгоритмів. Студенту потрібно продемонструвати глибоке розуміння ідеї фазового підходу та коректно реалізувати взаємодію між BFS та DFS для пошуку множини шляхів.

5. Порядок виконання

- Представлення графа: Граф зручно представляти списками суміжності. Оскільки граф дводольний, можна зберігати списки лише для однієї долі (напр., U).
- Паросполучення: Паросполучення зручно зберігати у вигляді двох масивів/словників `matchU` і `matchW`, де `matchU[u]=w` і `matchW[w]=u`.
- Реалізація BFS: Напишіть функцію, що приймає граф та поточне паросполучення, і обчислює масив відстаней `dist` для вершин. Запускати BFS потрібно відразу з усіх вільних вершин долі U.
- Реалізація DFS: Напишіть рекурсивну функцію, яка рухається по шарах, визначених BFS. Вона повинна оновлювати паросполучення, коли знаходить шлях.
- Головний цикл: Організуйте цикл `while`, який керує Крок ми: викликає BFS, а потім, якщо шляхи знайдено, викликає DFS для кожної вільної вершини з U.
- Тестування: Підготуйте набір тестів, що включають графи різного розміру та структури. Перевірте коректність результатів на прикладах, де максимальне паросполучення відоме.

6. Результат

Результатом роботи є програма, що ефективно знаходить максимальне паросполучення в дводольному графі, та звіт, що пояснює реалізацію та демонструє її роботу.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
  - Функція BFS для побудови шарів.
  - Функція DFS для пошуку шляхів.

    - Головний цикл алгоритму.
- Продемонструвати роботу додатку:
    - На кількох прикладах дводольних графів.
    - Пояснити на невеликому прикладі, як проходять перші 1-2 фази алгоритму.
- Бути готовим відповісти на питання:
    - Питання з розділу “Контрольні запитання”.
    - Як ваша реалізація BFS відрізняється від стандартного BFS?
    - Чому ваш DFS гарантовано знаходить саме найкоротші шляхи?
    - Як би змінилася складність, якби ви шукали лише один шлях за фазу?

Тести

1. Алгоритм Гопкрофта-Карпа призначений для пошуку максимального паросполучення в:

a) Будь-яких графах

b) Зважених дводольних графах

c) Незважених дводольних графах

d) Орієнтованих ациклічних графах

2. Що алгоритм Гопкрофта-Карпа намагається знайти на кожній фазі?

a) Один доповнюючий шлях

b) Максимальний набір вершинно-неперетинних найкоротших доповнюючих шляхів

c) Шлях, що проходить через усі вершини

d) Мінімальний розріз графа

3. Яка часова складність алгоритму Гопкрофта-Карпа?

a) $O(E \cdot V)$

b) $O(V^3)$

c) $O(E\sqrt{V})$

d) $O(E+V)$

4. “Перевертання” доповнюючого шляху призводить до:

a) Збільшення розміру паросполучення на 1

b) Зменшення розміру паросполучення на 1

c) Збільшення розміру паросполучення на 2

d) Не змінює розмір паросполучення

5. Який з алгоритмів за своєю ідеологією (фазовий підхід, BFS+DFS) найбільш схожий на алгоритм Гопкрофта-Карпа?

a) Алгоритм Прима

b) Алгоритм Дейкстри

c) Алгоритм Крускала

d) Алгоритм Дініца

Відповіді на тести

1. - c)
2. - b)
3. - c)
4. - a)
5. - d)

Тести підвищеної складності

1. Ключова теоретична властивість, яка забезпечує складність $O(E\sqrt{V})$, це те, що:

a) Кількість ітерацій DFS обмежена.

b) Кількість фаз алгоритму не перевищує $O(\sqrt{V})$.

c) BFS працює за час $O(\sqrt{V})$.

d) Граф стає зв'язним після першої фази.

2. У чому полягає основна відмінність BFS у алгоритмі Гопкрофта-Карпа від стандартного BFS?

a) Він використовує стек замість черги.

b) Він рухається по черзі по ребрах з паросполучення і не з паросполучення.

c) Він працює лише на ациклічних графах.

d) Він знаходить найважчий шлях.

3. Якщо після фази BFS виявилось, що `dist[NIL] == ∞`, це означає, що:

a) В графі є ізольовані вершини.

b) Поточне паросполучення є максимальним.

c) Сталася помилка в реалізації BFS.

d) Потрібно запустити DFS з усіх вершин.

4. Якщо звести задачу про максимальне паросполучення до задачі про максимальний потік і розв'язати її алгоритмом Дініца, яка буде складність?

   a) $O(V^2E)$

   b) $O(E\sqrt{V})$

   c) $O(VE)$

   d) $O(V^3)$

5. Алгоритм Гопкрофта-Карпа гарантує знаходження максимального набору вершинно-неперетинних шляхів, тому що:

   a) DFS запускається лише один раз.

   b) DFS позначає вершини як відвідані в межах однієї фази і не використовує їх повторно.

c) BFS гарантує, що шляхи не перетинаються.

d) Це властивість дводольних графів.

6. Якщо граф не є дводольним, алгоритм Гопкрофта-Карпа:

a) Все одно працюватиме коректно.

b) Видасть помилку під час виконання.

c) Може повернути не максимальне паросполучення або працювати некоректно.

d) Працюватиме довше, але знайде правильну відповідь.

7. “Теорема Берже” стверджує, що паросполучення є максимальним, якщо:

a) Всі вершини покриті паросполученням.

b) Немає жодного ребра, яке можна додати до паросполучення.

c) Немає доповнюючого шляху.

d) Розмір паросполучення дорівнює $|V|/2$.

8. Яка головна причина того, що довжина найкоротшого доповнюючого шляху строго зростає з кожною фазою?

a) Алгоритм запам’ятовує всі попередні шляхи.

b) Після перевертання всіх найкоротших шляхів довжини k, будь-який новий доповнюючий шлях мусить використовувати “обхідні” ребра.

c) Кількість вільних вершин зменшується.

d) Це випадковий процес.

9. Чи можна використовувати алгоритм Гопкрофта-Карпа для пошуку максимального зваженого паросполучення?

a) Так, якщо ваги є цілими числами.

b) Так, але складність зросте.

c) Ні, алгоритм не враховує ваги ребер.

d) Так, якщо попередньо відсортувати ребра за вагою.

10. Якщо після роботи алгоритму залишились вільні вершини, це означає, що:

a) Алгоритм працює неправильно.

b) Не існує досконалого паросполучення (яке покриває всі вершини).

c) Потрібно запустити ще одну фазу.

d) Це неможливо, в максимальному паросполученні не буває вільних вершин.

Відповіді на тести підвищеної складності

1. - b) Це ключовий теоретичний результат, що лежить в основі аналізу складності алгоритму.

2. - b) BFS у цьому алгоритмі є багатоджерельним (починається з усіх вільних вершин) і рухається по черзі по ребрах з паросполучення та не з нього, будуючи шари чергувальних шляхів.

3. - b) Якщо після фази BFS не знайдено шляху до жодної вільної вершини в іншій долі, це означає, що доповнюючих шляхів більше не існує, а отже, поточне паросполучення є максимальним.

4. - b) Зведення до потоку створює одиничну мережу. Складність алгоритму Дініца для таких мереж становить $O(E\sqrt{V})$, що ідентично складності Гопкрофта-Карпа.

5. - b) DFS, що запускається після BFS, рухається по шарах. Позначаючи вершини як відвідані в межах однієї великої сесії DFS (для однієї фази), ми гарантуємо, що вони не будуть використані в іншому шляху цього ж набору.

6. - c) Алгоритм суттєво покладається на структуру дводольного графа (чергування між долями U і W). Наявність циклів непарної довжини порушить його логіку.

7. - c) Це формулювання теореми Берже, яка є теоретичною основою для всіх алгоритмів, що базуються на пошуку доповнюючих шляхів.

8. - b) Коли всі найкоротші шляхи довжини k використані" (перевернуті), симетрична різниця між новим та старим паросполученням не містить

доповнюючих шляхів довжини k. Будь-який новий шлях змушений буде обходити використані вершини, що робить його довшим.

9. - c) Алгоритм Гопкрофта-Карпа працює з наявністю або відсутністю ребер, але не з їх вагами. Для зважених графів потрібні інші алгоритми, наприклад, Угорський алгоритм.

10. - b) Максимальне паросполучення не завжди є досконалим. Якщо в графі неможливо підібрати пару для кожної вершини, то в будь-якому максимальному паросполученні залишаться вільні вершини.

Завдання для самостійної роботи

- Реалізуйте простий алгоритм пошуку максимального паросполучення, що знаходить один доповнюючий шлях за раз за допомогою DFS. Порівняйте його продуктивність з алгоритмом Гопкрофта-Карпа.
- Напишіть програму, яка перевіряє, чи є заданий граф дводольним. Якщо так, вона знаходить у ньому максимальне паросполучення.
- Візуалізуйте роботу алгоритму Гопкрофта-Карпа: показуйте граф, поточне паросполучення, а на кожній фазі — шари, побудовані BFS, та шляхи, знайдені DFS.
- Знайдіть зв’язок між розміром максимального паросполучення та мінімальним вершинним покриттям у дводольному графі (Теорема Кеніга). Напишіть програму для пошуку обох.
- Адаптуйте алгоритм для розв’язання задачі: є група людей і набір завдань, кожна людина може виконувати певний набір завдань. Потрібно призначити якомога більше людей на унікальні завдання.
- Дослідіть, як змінюється кількість фаз алгоритму залежно від щільності графа.
- Напишіть програму для пошуку досконалого паросполучення (паросполучення, що покриває всі вершини графа).
- Реалізуйте алгоритм Гопкрофта-Карпа, використовуючи ітеративний BFS та ітеративний DFS (з використанням стека).

– Прочитайте про Угорський алгоритм для пошуку максимального зваженого паросполучення і напишіть есей, що порівнює його ідеї з ідеями алгоритму Гопкрофта-Карпа.

– Розробіть стрес-тест, що генерує складні випадки для алгоритму Гопкрофта-Карпа.

Теми лабораторних робіт

1. Реалізація алгоритму Гопкрофта-Карпа та порівняльний аналіз продуктивності з методом на основі максимального потоку.
2. Розробка системи для автоматичного складання розкладів (напр., пари-аудиторії) з використанням моделі дводольного графа.
3. Створення інтерактивного візуалізатора для покрокової демонстрації роботи алгоритму Гопкрофта-Карпа.
4. Застосування алгоритму для вирішення задач в онлайн-системах знайомств (рекомендація пар).
5. Реалізація та аналіз алгоритму для пошуку мінімального вершинного покриття в дводольних графах.
6. Розробка програми для аналізу ринку праці: зіставлення кандидатів з вакансіями на основі навичок.
7. Порівняння ефективності алгоритму Гопкрофта-Карпа на щільних та розріджених дводольних графах.
8. Розробка модуля для системи планування ресурсів, що оптимізує призначення завдань на основі максимального паросполучення.
9. Реалізація алгоритму пошуку всіх максимальних паросполучень у графі.
10. Аналіз стабільності паросполучень та порівняння з задачею про максимальне паросполучення.

Паросполучення в загальних графах. Алгоритм Блоссом

У попередньому розділі ми розглянули високоефективний алгоритм Гопкрофта-Карпа, який чудово справляється із пошуком максимального

паросполучення у дводольних графах. Але що робити, якщо граф не є дводольним? Чи можемо ми застосувати ту саму логіку пошуку доповнюючих шляхів?

Виявляється, що пряме застосування цих методів не працює. Причина криється у фундаментальній відмінності загальних графів від дводольних: наявності циклів непарної довжини. Саме ці цикли створюють ситуації, з якими прості алгоритми пошуку чергувальних шляхів не можуть впоратися.

Алгоритм Блоссом, розроблений Джеком Едмондсом, був справжнім проривом. Він став першим поліноміальним алгоритмом для пошуку максимального паросполучення в будь-яких (загальних) графах. Його центральна ідея — геніальний спосіб обробки циклів непарної довжини шляхом їх "стиснення" в одну вершину.

1. Теоретичні відомості

1.1. Проблема циклів непарної довжини

Давайте розглянемо, чому цикли непарної довжини є проблемою. Уявіть, що ми будуємо чергувальне дерево (дерево з чергувальних шляхів), починаючи з вільної вершини. У дводольному графі, коли ми знаходимо ребро між двома вершинами, що вже є в дереві, вони завжди знаходяться на різних "рівнях" (одна на парній відстані від кореня, інша — на непарній). Це утворює доповнюючий шлях або цикл парної довжини, що не створює проблем.

Але в загальному графі ми можемо знайти ребро між двома вершинами, що знаходяться на однаковому рівні в чергувальному дереві. Це утворює цикл непарної довжини. Такий цикл не є доповнюючим шляхом і заважає простому пошуку.

1.2. Квітки та їх властивості

Квітка (Blossom) — це цикл непарної довжини в графі, в якому рівно $\frac{|C|-1}{2}$ ребер належать до поточного паросполучення M, де $|C|$ — кількість вершин у циклі. Вершина в циклі, яка найближча до кореня чергувального дерева, називається базою квітки. Шлях від кореня дерева до бази є чергувальним.

Головна ідея алгоритму полягає в наступному: якщо ми знайшли таку квітку, ми можемо стиснути її в одну "супер-вершину" і продовжити пошук доповнюючого шляху в цьому новому, меншому графі.

Чому це працює? Едмондс довів ключову теорему: доповнюючий шлях існує в оригінальному графі G тоді і тільки тоді, коли він існує в стиснутому графі

G'. Це означає, що операція стиснення є “безпечною” — ми не втрачаємо інформацію про існування доповнюючого шляху.

1.3. Основні операції алгоритму

Алгоритм базується на ітеративному пошуку доповнюючих шляхів для кожної вільної вершини. Під час цього пошуку він може виконувати дві унікальні операції:

1. Стиснення квітки

– Коли відбувається: Під час побудови чергувального дерева з коренем у вільній вершині r, ми знаходимо ребро (u, v), де u і v вже знаходяться в дереві на однаковій відстані від r.

– Дія:

  – Знаходимо цикл непарної довжини, утворений цим ребром та шляхами від u і v до їхнього найближчого спільного предка (бази квітки).

  – Створюємо новий граф G', в якому всі вершини квітки “зливаються” в одну супер-вершину. Усі ребра, інцидентні вершинам квітки, тепер стають інцидентними цій супер-вершині.

  – Продовжуємо пошук доповнюючого шляху в G'. Цей процес може бути рекурсивним — у стиснутому графі ми можемо знайти нову квітку і стиснути її знову.

2. Розширення квітки

– Коли відбувається: Коли ми знайшли доповнюючий шлях у стиснутому графі G', і цей шлях проходить через супер-вершину, що представляє квітку.

– Дія:

  – Нам потрібно “підняти” цей шлях назад до оригінального графа G.

  – Ми входимо в квітку через певну вершину. Всередині квітки ми можемо вибрати один з двох шляхів, щоб вийти з неї, зберігши при цьому чергувальну властивість всього шляху.

- Вибір правильного шляху всередині квітки гарантує, що ми отримаємо коректний доповнюючий шлях в оригінальному графі.

1.4. Загальна структура алгоритму

Алгоритм можна описати наступним чином:

- Ініціалізуємо порожнє паросполучення M.
- Для кожної вільної вершини v в графі:
  - Намагаємося знайти доповнюючий шлях, що починається з v. Для цього будуємо чергувальне дерево.
  - Під час побудови дерева, якщо зустрічаємо ребро (u, w):
    - Якщо w — невідвідана вершина, розширюємо дерево.
    - Якщо w — вершина в тому ж дереві на тому ж рівні, ми знайшли квітку. Стискаємо її та продовжуємо пошук у новому графі (рекурсивно).
    - Якщо w — вершина в іншому чергувальному дереві (побудованому з іншої вільної вершини), ми знайшли доповнюючий шлях.
  - Якщо доповнюючий шлях P знайдено:
    - Якщо P проходить через стиснуті квітки, розширюємо їх, щоб отримати шлях в оригінальному графі.
    - “Перевертаємо” шлях P, збільшуючи розмір паросполучення на 1.
    - Переходимо до наступної вільної вершини.
- Якщо ми перебрали всі вільні вершини і не знайшли доповнюючих шляхів, поточне паросполучення є максимальним.

Псевдокод такий:

Через виняткову складність повної реалізації, наведемо високорівневий псевдокод, що відображає основну логіку.

```
function BlossomAlgorithm(G):
  M = {} // Порожнє паросполучення
```

```
  while exists a free vertex v in G:
    P = Find_Augmenting_Path_With_Blossoms(G, M, v)
    if P exists:
      M = M XOR P // "Перевертаємо" шлях P
    else:
      // Позначаємо v, щоб не починати з неї знову
      mark v as processed

  return M

function Find_Augmenting_Path_With_Blossoms(G, M, r):
  // Будуємо чергувальне дерево з коренем r
  forest = {r: (parent=NULL, level=0)}
  queue = {r}

  while queue is not empty:
    u = queue.dequeue()

    for each neighbor v of u:
      if (u,v) is in M: continue // Рухаємось лише по ребрах не з M

      if v is not in forest:
        // Розширюємо дерево
        w = matching_partner_of(v, M)
        add v and w to forest, queue
      else:
        if level of v in forest is even: // Знайшли ребро всередині о
дного дерева
          // Знайдено квітку
          B = find_blossom(u, v, forest)
          G_shrunk = shrink_graph(G, B)
          M_shrunk = shrink_matching(M, B)

          // Рекурсивний виклик
          P_shrunk = Find_Augmenting_Path_With_Blossoms(G_shrunk, M_s
hrunk, r_shrunk)

          if P_shrunk exists:
            P = expand_path(P_shrunk, B)
            return P

  // Якщо обійшли все дерево і не знайшли шлях
  return NULL
```

1.5. Складність

Перша реалізація алгоритму мала складність $O(V^4)$. Згодом її було покращено до $O(V^3)$, а найефективніша відома реалізація, запропонована Габовим, має складність $O(E \cdot V)$. Це робить алгоритм ефективним для розв'язання задачі на графах загального вигляду.

2. Область застосування

Алгоритм Блоссом є фундаментальним для будь-якої задачі, що вимагає пошуку максимального паросполучення в графі, який не є гарантовано дводольним.

- Загальні задачі про призначення: Коли обмеження є симетричними (якщо А може працювати з Б, то і Б може з А), граф може містити цикли непарної довжини.
- Обчислювальна хімія: Аналіз молекулярних структур, де атоми — вершини, а хімічні зв'язки — ребра.
- Статистична фізика: Моделювання взаємодій у кристалічних решітках.
- Оптимізація маршрутів: У деяких задачах про планування маршрутів, де потрібно спарувати точки візиту.
- Теоретична інформатика: Як важливий приклад потужного поліноміального алгоритму для NP-складної задачі на певних підкласах графів.

3. Приклад застосування

Уявімо граф-п'ятикутник $v_1 - v_2 - v_3 - v_4 - v_5 - v_1$ і "хвіст" $v_5 - v_6$, де v_6 — вільна вершина.

- Початкове паросполучення: $M = (v_1, v_2), (v_3, v_4)$. Вершини v_5, v_6 — вільні.

Пошук доповнюючого шляху з v_6:

1. Починаємо з v_6: Будуємо чергувальне дерево.
   - v_6 на рівні 0.
   - Йдемо по ребру (v_6, v_5), якого немає в M. v_5 на рівні 1.
   - Вершина v_5 вільна! Знайдено доповнюючий шлях $v_6 - v_5$.
2. Перевертаємо шлях:

- Додаємо (v_6, v_5) до M.
- Нове паросполучення: $M = (v_1, v_2), (v_3, v_4), (v_5, v_6)$. Розмір 3. Це максимальне паросполучення.

Складніший приклад з квіткою:

- Граф: п'ятикутник $v_1 - v_2 - v_3 - v_4 - v_5 - v_1$.
- Початкове паросполучення: $M = (v_2, v_3)$. Вільні вершини: v_1, v_4, v_5.

Пошук доповнюючого шляху з v_1:

3. Будуємо дерево:
   - v_1 (корінь) на рівні 0.
   - Йдемо по (v_1, v_2) (не в M) $\Longrightarrow v_2$ на рівні 1.
   - Йдемо по (v_2, v_3) (в M) $\Longrightarrow v_3$ на рівні 2.
   - Водночас з v_1 йдемо по (v_1, v_5) (не в M) $\Longrightarrow v_5$ на рівні 1.
4. Знаходимо квітку:
   - З v_3 (рівень 2) є ребро до v_4 (не в M). Припустимо, v_4 ще не в дереві. Тоді v_4 на рівні 3.
   - З v_4 є ребро до v_5. Але v_5 вже в дереві на рівні 1!
   - Оскільки ребро (v_4, v_5) з'єднує вершини на рівнях 3 і 1, це не квітка. Це доповнюючий шлях! $v_1 \to v_2 \to v_3 \to v_4 \to v_5$. Ні, шлях має бути чергувальним!
   - Давайте знову: з v_3 (рівень 2) є ребро (v_3, v_4). v_4 не спарована. $v_1 \to v_2 \to v_3 \to v_4$ є доповнюючим шляхом. Перевертаємо його. Нове паросполучення $M' = (v_1, v_2), (v_3, v_4)$.

Давайте знайдемо квітку. Припустимо, ми шукаємо шлях з v_1, і дерево розрослося так, що v_3 і v_5 знаходяться на рівні 2. І ми знаходимо ребро (v_3, v_4), а потім (v_4, v_5). Це складно проілюструвати текстом. Ключовий момент: квітка знаходиться, коли ребро з'єднує дві вершини на парній відстані від кореня в чергувальному дереві.

- Стиснення: Цикл (квітка) стискається в одну супер-вершину B.
- Пошук у G': Шукаємо доповнюючий шлях у новому графі.

– Розширення: Якщо шлях проходить через B, ми “розкриваємо” квітку і прокладаємо шлях через її вершини.

4. Переваги та недоліки

Переваги

– Загальність: Це перший і найвідоміший поліноміальний алгоритм, що розв’язує задачу для будь-яких графів, а не лише для дводольних.

– Теоретична краса: Ідея стиснення та розширення квіток є надзвичайно елегантною і важливою в теорії алгоритмів.

Недоліки

– Надзвичайна складність реалізації: Алгоритм Блоссом вважається одним з найскладніших для коректного впровадження класичних графових алгоритмів. Він вимагає складних структур даних та ретельного опрацювання багатьох випадків.

– Повільніший за спеціалізовані аналоги: На дводольних графах він значно поступається за швидкістю алгоритму Гопкрофта-Карпа.

– Існують швидші реалізації: Хоча класична складність $O(V^3)$, сучасніші алгоритми (напр., алгоритм Мікалі та Вазірані) досягають тієї ж складності $O(E\sqrt{V})$, що й Гопкрофт-Карп, але є ще набагато складнішими.

Контрольні запитання

– Чому алгоритми для дводольних графів не працюють на загальних графах? Яка структурна особливість є перешкодою?

– Що таке “квітка” (blossom) в контексті цього алгоритму?

– Поясніть своїми словами ідею операції “стиснення квітки”. Чому ця операція є коректною?

– В яких випадках потрібно виконувати операцію “розширення квітки”?

– Яку роль відіграє поняття “бази” квітки?

– Чому алгоритм Блоссом вважається важливим теоретичним проривом?

Методичні вказівки до лабораторної роботи

1. Назва

Дослідження алгоритму Блоссом для пошуку максимального паросполучення.

2. Мета

- Зрозуміти фундаментальні перешкоди для пошуку паросполучення в загальних графах.
- Вивчити концепцію “квіток” та ідею їх стиснення.
- Реалізувати окремі компоненти алгоритму Блоссом.

3. Завдання

- Реалізувати структуру даних для загального графа та представлення паросполучення.
- Реалізувати функцію, яка будує чергувальне дерево від заданої вільної вершини.
- Реалізувати функцію, яка знаходить цикли непарної довжини (“квітки”) в цьому дереві.
- (Основне завдання) Реалізувати функцію `ShrinkBlossom`, яка приймає на вхід граф та набір вершин квітки, і повертає новий, стиснутий граф.
- Продемонструвати роботу цих компонентів на прикладах.

4. Опис

Повна реалізація алгоритму Блоссом є надзвичайно складною і виходить за рамки стандартної лабораторної роботи. Тому пропонується зосередитись на реалізації та розумінні його ключових компонентів. Студент повинен продемонструвати, як знаходиться квітка і як відбувається операція її стиснення.

5. Порядок виконання

- Проектування: Сплануйте, як ви будете представляти граф, паросполучення, чергувальне дерево (напр., через масив батьківських вершин) та стиснутий граф.

- Пошук шляху/дерева: Реалізуйте BFS-подібний пошук для побудови чергувального дерева. Зберігайте для кожної вершини її батька в дереві та її "рівень" (парний/непарний).
- Детектор квіток: Під час пошуку, коли ви розглядаєте ребро (u, v) і знаходите, що v вже в дереві, перевірте рівні u і v. Якщо вони однакові, ви знайшли квітку.
- Реалізація стиснення: Напишіть функцію, яка створює новий граф. Всі вершини квітки замінюються однією новою вершиною. Ребра, що були інцидентні вершинам квітки, тепер приєднуються до нової супер-вершини.
- Тестування: Створіть кілька графів з циклами непарної довжини і продемонструйте, що ваша програма коректно знаходить квітки і виконує їх стиснення.

6. Результат

Результатом роботи є програмний код, що реалізує ключові етапи алгоритму Блоссом (пошук квіток, стиснення), та звіт, що пояснює ці концепції та демонструє їх роботу на прикладах.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
  - Функція побудови чергувального дерева.
  - Функція детектування квіток.
  - Функція стиснення графа.
- Продемонструвати роботу додатку:
  - На графі, що не має циклів непарної довжини.
  - На графі, що містить одну або кілька "квіток", і показати результат стиснення.
- Бути готовим відповісти на питання:
  - Питання з розділу "Контрольні запитання".
  - Як ви знаходите "базу" квітки?

– Поясніть, як би ви реалізували операцію “розширення” квітки.

– Чому цей алгоритм не потрібен для дводольних графів?

Тести

1. Основною перешкодою для простих алгоритмів пошуку паросполучення в загальних графах є:

a) Цикли парної довжини

b) Цикли непарної довжини

c) Ізольовані вершини

d) Повні підграфи

2. Алгоритм Блоссом отримав свою назву через:

a) Схожість графа на квітку

b) Прізвище його автора

c) Назву структури даних, яку він використовує

d) Специфічну структуру (цикл непарної довжини), яку він знаходить і стискає

3. Операція “стиснення квітки” полягає у:

a) Видаленні циклу непарної довжини з графа

b) Заміні циклу непарної довжини на одну супер-вершину

c) Знаходженні найкоротшого шляху всередині циклу

d) Додаванні нових ребер для усунення циклу

4. Алгоритм Блоссом був першим відомим…

a) Експоненційним алгоритмом для паросполучень

b) Поліноміальним алгоритмом для паросполучень у загальних графах

c) Алгоритмом для зважених паросполучень

d) Квантовим алгоритмом для графів

5. Після знаходження доповнюючого шляху в стиснутому графі, який проходить через супер-вершину, необхідно:

a) Проігнорувати цей шлях

b) Видалити супер-вершину

c) Виконати операцію “розширення квітки”

d) Запустити алгоритм з початку

Відповіді на тести

1. - b)

2. - d)

3. - b)

4. - b)

5. - c)

Тести підвищеної складності

1. Чому операція стиснення квітки є коректною (зберігає властивість існування доповнюючого шляху)?

a) Тому що будь-який шлях, що входив у квітку, може з неї вийти, зберігши чергування.

b) Це просто евристика, яка добре працює.

c) Тому що квітка завжди містить непарну кількість ребер.

d) Тому що після стиснення граф стає дводольним.

2. “База” квітки — це:

a) Будь-яка вершина в циклі.

b) Вершина в циклі, яка була відвідана першою.

c) Найближчий спільний предок у чергувальному дереві для двох вершин, що утворюють квітку.

d) Вершина з найбільшим степенем у циклі.

3. Яка сучасна оцінка складності для пошуку максимального паросполучення в загальних графах, досягнута складнішими алгоритмами?

a) $O(E + V)$

b) $O(E\log V)$

c) $O(E\sqrt{V})$

d) $O(V^3)$

4. Якщо під час пошуку шляху з вільної вершини r ми знаходимо ребро до вершини v, що належить чергувальному дереву, побудованому з іншої вільної вершини s, то ми знайшли:

a) Квітку

b) Міст

c) Доповнюючий шлях

d) Тупикову гілку

5. Що робить реалізацію алгоритму Блоссом особливо складною?

a) Необхідність використовувати числа з рухомою комою.

b) Великі вимоги до пам’яті.

c) Необхідність керувати рекурсивними стисненнями та розширеннями графів і паросполучень.

d) Висока асимптотична складність.

6. Якщо застосувати алгоритм Блоссом до дводольного графа, що станеться?

a) Він видасть помилку.

b) Він ніколи не знайде квітку і працюватиме як простіший алгоритм пошуку шляхів.

c) Він працюватиме швидше, ніж Гопкрофт-Карп.

d) Він знайде неправильну відповідь.

7. Чи можна використати ідеї алгоритму Блоссом для пошуку максимального *зваженого* паросполучення?

a) Ні, ідеї абсолютно несумісні.

b) Так, існують складніші алгоритми для зваженого випадку, які використовують ідею стиснення квіток.

c) Так, достатньо просто додати ваги ребер.

d) Ні, тому що ваги порушують структуру чергувальних шляхів.

8. Яка структура даних найкраще підходить для представлення чергувального дерева, що будується під час пошуку?

a) Масив батьківських вказівників та масив рівнів.

b) Матриця суміжності.

c) Система неперетинних множин (DSU).

d) Черга з пріоритетом.

9. У фінальному паросполученні, знайденому алгоритмом, може не бути жодного ребра з початкової квітки, яка була стиснута.

a) Правда.

b) Неправда.

c) Це можливо, лише якщо квітка складалася з 3 вершин.

d) Це можливо, лише якщо паросполучення не є максимальним.

10. “Найближчий спільний предок” (LCA) в чергувальному дереві потрібен для того, щоб:

a) Визначити розмір паросполучення.

b) Прискорити BFS.

c) Точно визначити вершини, що входять до складу квітки.

d) Вибрати наступну вільну вершину для обробки.

Відповіді на тести підвищеної складності

1. - a) Стиснення зберігає інформацію про парність довжини шляху до будь-якої вершини поза квіткою. Якщо існує доповнюючий шлях, він або не проходить через квітку, або проходить, і в обох випадках його можна знайти в стиснутому графі.

2. - c) Це вершина циклу, яка є предком для всіх інших вершин циклу в чергувальному дереві. Саме від неї розходяться два шляхи, що утворюють квітку.

3. - c) Алгоритм, розроблений Мікалі та Вазірані, досягає складності $O(E\sqrt{V})$, що є найкращою відомою на сьогодні.

4. - c) Це класичний випадок знаходження доповнюючого шляху, що з'єднує два окремі чергувальні дерева.

5. - c) Основна складність полягає в коректному управлінні даними при рекурсивних викликах: потрібно правильно передавати і модифікувати граф, парасполучення та шляхи на різних рівнях стиснення.

6. - b) Алгоритм коректно обробить дводольний граф. Він просто ніколи не знайде умову для створення квітки (ребро між вершинами на однаковому парному рівні) і буде працювати як повільніший аналог алгоритму Гопкрофта-Карпа.

7. - b) Так, загальний алгоритм для зваженого випадку, розроблений Едмондсом, є розширенням алгоритму Блоссом і використовує схожі ідеї стиснення циклів.

8. - a) Масив батьків дозволяє відновлювати шляхи до кореня, а масив рівнів (або відстаней) потрібен для швидкої перевірки умов утворення квітки.

9. - a) Правда. Під час розширення квітки доповнюючий шлях може бути прокладений через квітку таким чином, що ребро з початкового парасполучення всередині квітки буде "викинуто" з нового парасполучення, а натомість будуть використані два інших ребра з квітки.

10. - c) Коли пошук знаходить ребро між вершинами u і v, які вже в дереві, нам потрібно знайти їх шляхи назад до кореня. Точка, де ці шляхи сходяться (LCA), є базою квітки, а самі шляхи разом з ребром (u, v) утворюють цикл.

Завдання для самостійної роботи

1. Намалюйте граф з 7 вершин, що містить цикл довжиною 5. Простежте вручну, як алгоритм Блоссом знайде в ньому максимальне паросполучення.
2. Реалізуйте функцію, яка перевіряє, чи є заданий граф дводольним.
3. Реалізуйте функцію `FindLCA` (пошук найближчого спільного предка) для дерева, представленого через масив батьківських вказівників.
4. Напишіть програму, яка знаходить у графі всі цикли непарної довжини.
5. Прочитайте про “Теорему Татта”, яка дає умову існування досконалого паросполучення в загальному графі. Порівняйте її з “Теоремою Холла” для дводольних графів.
6. Реалізуйте операцію `ExpandBlossom`. Припустімо, у вас є доповнюючий шлях у стиснутому графі. Напишіть функцію, що перетворює його на шлях в оригінальному графі.
7. Дослідіть, як можна використати задачу про максимальне паросполучення для розв’язання задачі про мінімальне реберне покриття.
8. Створіть невелику бібліотеку, яка містить алгоритм пошуку паросполучення для дводольних графів (напр., на основі DFS) та реалізовані вами компоненти алгоритму Блоссом.
9. Напишіть есей на тему “Чому цикли непарної довжини є ключовою проблемою в теорії паросполучень?”.
10. Знайдіть та опишіть реальний приклад задачі, яка моделюється недводольним графом і вимагає пошуку максимального паросполучення.

Теми лабораторних робіт

1. Реалізація ключових компонентів алгоритму Блоссом: пошук та стиснення квіток.
2. Порівняльний аналіз простого алгоритму пошуку паросполучення на основі DFS та його поведінки на дводольних і недводольних графах.

3. Створення інтерактивного візуалізатора, що демонструє знаходження квітки та її стиснення.
4. Розробка програми, що визначає тип графа (дводольний/загальний) і вибирає відповідний метод для пошуку максимального паросполучення.
5. Застосування алгоритму Блоссом для аналізу структур даних в обчислювальній хімії.
6. Дослідження та реалізація алгоритму Габова для пошуку паросполучення з кращою асимптотикою.
7. Задача про мінімальне реберне покриття в загальних графах та її зв'язок з максимальним паросполученням.
8. Розробка модуля для системи автоматизованого проектування, що використовує паросполучення для оптимізації з'єднань.
9. Реалізація алгоритму для пошуку максимального зваженого паросполучення в загальних графах (алгоритм Едмондса).
10. Порівняння підходів на основі паросполучень та на основі потоків для вирішення одних і тих самих задач.

Система неперетинних множин

У розділі, присвяченому алгоритму Крускала, ми вже познайомилися з елегантною та потужною структурою даних — системою неперетинних множин. Ми бачили, як вона дозволила нам ефективно визначати, чи не утворить нове ребро цикл, шляхом перевірки, чи належать дві вершини до одного компонента зв'язності. Тоді ми лише поверхнево торкнулися її реалізації. Тепер настав час зануритися глибше і дослідити, що робить цю структуру даних однією з найефективніших у комп'ютерних науках.

Наївна реалізація DSU, де ми представляємо множини як дерева і довільно приєднуємо одне до іншого, може призвести до серйозних проблем з продуктивністю. У найгіршому випадку наші "дерева" можуть виродитися в довгі ланцюжки, і тоді операція `Find` (пошук представника множини)

займатиме лінійний час $O(N)$, що є неприйнятно повільним для великих даних.

Саме тут на сцену виходять дві прості, але надзвичайно потужні оптимізації, які разом зводять середній час виконання операцій до майже константної величини:

- Об'єднання за рангом/розміром: Це проста евристика, яка керує процесом об'єднання множин. Замість того, щоб діяти навмання, ми завжди приєднуємо менше дерево до кореня більшого. Цей простий трюк не дозволяє деревам ставати надто високими і "приземкуватими".
- Стиснення шляху: Ця оптимізація перетворює операцію `Find` з пасивного спостерігача на активного оптимізатора. Кожного разу, коли ми шукаємо корінь для певної вершини, ми "сплющуємо" дерево, змушуючи всі вершини на пройденому шляху вказувати безпосередньо на корінь. Це різко прискорює всі майбутні операції `Find` для цих вершин.

Поєднання цих двох технік створює структуру даних, чия амортизована складність операцій дорівнює $O(\alpha(N))$, де $\alpha(N)$ — обернена функція Аккермана. Ця функція росте настільки повільно, що для будь-яких практично мислимих значень N її значення не перевищує 5.

У цьому розділі ми детально розберемо кожну з цих оптимізацій, проаналізуємо їхній вплив і навчимося реалізовувати одну з найшвидших структур даних у вашому арсеналі.

Оптимізація "Об'єднання за рангом"

У попередньому вступному розділі ми встановили, що наївна реалізація системи неперетинних множин (DSU) є вразливою. Послідовність невдалих операцій `Union` може перетворити наші дерева на довгі ланцюжки, що деградують продуктивність операції `Find` до лінійного часу $O(N)$. Це зводить нанівець усі переваги використання DSU.

Перша з двох ключових оптимізацій, яку ми розглянемо, — об'єднання за рангом. Це проста, але глибока евристика, що керує тим, як ми об'єднуємо дерева. Замість того, щоб діяти наосліп, ми будемо дотримуватися простого

правила, яке не дозволить нашим деревам ставати надто “високими” і “тонкими”.

1. Теоретичні відомості

1.1. Проблема наївної реалізації `Union`

Давайте наочно продемонструємо проблему. Уявіть, що ми послідовно виконуємо операції `Union(1, 2)`, `Union(2, 3)`, `Union(3, 4)` і так далі. Якщо наша наївна реалізація `Union(u, v)` завжди приєднує дерево `v` до кореня дерева `u`, ми отримаємо таку структуру:

- `Union(1, 2)`: `parent[2] = 1`.
- `Union(2, 3)`: `parent[3] = 1`. (корінь 2 це 1)
- `Union(3, 4)`: `parent[4] = 1`. (корінь 3 це 1) Це виглядає добре. А якщо `Union(u, v)` приєднує корінь `u` до кореня `v`?
- `Union(1, 2)`: `parent[1] = 2`.
- `Union(2, 3)`: `parent[2] = 3`.
- `Union(3, 4)`: `parent[3] = 4`. В результаті ми отримуємо ланцюжок $1 \to 2 \to 3 \to 4$, де 4 є коренем. Тепер, щоб виконати `Find(1)`, нам доведеться пройти весь цей ланцюжок, що займає $O(N)$ часу. Саме цієї деградації ми хочемо уникнути.

1.2. Ідея об’єднання за рангом

Ідея евристики полягає в тому, щоб завжди зберігати дерева якомога “приземкуватими”. Коли нам потрібно об’єднати два дерева, ми порівнюємо їхні висоти і приєднуємо коротше дерево до кореня вищого.

- Якщо ми приєднуємо дерево меншої висоти до дерева більшої висоти, загальна висота результуючого дерева не змінюється. Вона дорівнює висоті більшого з дерев.
- Висота результуючого дерева збільшується на одиницю тільки тоді, коли ми об’єднуємо два дерева однакової висоти.

Щоб реалізувати це, ми будемо зберігати для кожного кореня його ранг (rank). Ранг — це верхня межа висоти дерева. Ми використовуємо термін “ранг”, а не “висота”, тому що друга оптимізація (стиснення шляху) може зменшувати реальну висоту дерева, не оновлюючи це значення. Тому ранг — це “потенційна” висота.

1.3. Реалізація та псевдокод

Для реалізації нам знадобиться додатковий масив `rank` розміром $|V|$.

Операція `MakeSet`

При створенні нової множини, що складається з одного елемента, ми ініціалізуємо її ранг нулем.

```
function MakeSet(v, parent, rank):
  parent[v] = v
  rank[v] = 0
```

Операція `Find`

На даному етапі ми все ще використовуємо наївну версію `Find`, оскільки розглядаємо лише першу оптимізацію.

```
function Find(v, parent):
  if v == parent[v]:
    return v
  return Find(parent[v], parent)
```

Операція `Union` (з оптимізацією за рангом)

Це серце оптимізації. Тут ми реалізуємо нову логіку об'єднання.

```
function Union_by_Rank(u, v, parent, rank):
  // Знаходимо корені для обох елементів
  root_u = Find(u, parent)
  root_v = Find(v, parent)

  // Якщо елементи вже в одній множині, нічого не робимо
  if root_u == root_v:
    return

  // Порівнюємо ранги коренів
  if rank[root_u] < rank[root_v]:
    // Приєднуємо дерево з меншим рангом (u) до дерева з більшим ранг
ом (v)
    parent[root_u] = root_v
    // Ранг дерева v не змінюється
  else if rank[root_u] > rank[root_v]:
    // Приєднуємо дерево з меншим рангом (v) до дерева з більшим ранг
ом (u)
    parent[root_v] = root_u
    // Ранг дерева u не змінюється
  else:
    // Ранги однакові. Приєднуємо v до u і збільшуємо ранг u.
```

```
parent[root_v] = root_u
rank[root_u] = rank[root_u] + 1
```

1.4. Альтернатива: Об’єднання за розміром

Існує дуже схожа і настільки ж ефективна евристика — об’єднання за розміром. Замість рангів ми зберігаємо для кожного кореня кількість елементів у його піддереві (розмір множини).

- При об’єднанні ми завжди приєднуємо менше за розміром дерево до кореня більшого.
- Розмір нового кореня стає сумою розмірів двох об’єднаних дерев. Цей підхід також ефективно запобігає росту високих дерев і має таку ж асимптотичну складність. Іноді його легше реалізувати, оскільки не потрібно турбуватися про ранг, який є лише верхньою межею.

1.5. Аналіз складності

Застосування оптимізації об’єднання за рангом (або за розміром) кардинально покращує продуктивність.

- Ключова властивість: Дерево з рангом k містить щонайменше 2^k вершин.
    - *Доведення (індукцією):* База: дерево з рангом 0 має $1 = 2^0$ вершину. Крок: припустимо, твердження вірне для рангів менших за k. Дерево рангу k утворюється лише при злитті двох дерев рангу $k-1$. За припущенням, кожне з них мало щонайменше 2^{k-1} вершин. Отже, нове дерево має щонайменше $2^{k-1} + 2^{k-1} = 2 \cdot 2^{k-1} = 2^k$ вершин.
- Наслідок: Якщо в нас є N елементів, то максимальний ранг будь-якого дерева не може перевищувати $\log_2 N$. Чому? Бо якщо ранг дорівнює k, то в дереві щонайменше 2^k елементів. Отже, $2^k \leq N$, що еквівалентно $k \leq \log_2 N$.
- Часова складність: Оскільки висота дерева (а отже, і довжина шляху під час операції `Find`) обмежена рангом, то час виконання однієї операції `Find` становить $O(\log N)$. Операція `Union` виконує два виклики `Find` та декілька константних операцій, тому її складність також $O(\log N)$.

Це величезний стрибок у порівнянні з $O(N)$ для наївної реалізації. Ми перетворили повільну структуру даних на досить ефективну.

2. Область застосування

Об'єднання за рангом/розміром є обов'язковою частиною будь-якої практичної реалізації DSU. Без неї структура даних є надто повільною для більшості застосувань.

- Алгоритм Крускала: Ця оптимізація робить DSU достатньо швидкою для ефективної реалізації алгоритму Крускала, де загальна складність визначається сортуванням ребер, а не операціями з DSU.
- Аналіз зв'язності: У задачах, де потрібно динамічно об'єднувати компоненти зв'язності графа і перевіряти, чи належать дві вершини до однієї компоненти.
- Обробка зображень: Для сегментації зображень, де сусідні пікселі схожого кольору об'єднуються в одну множину.
- Будь-які задачі, що вимагають ефективного групування: У кластеризації, аналізі мереж тощо.

3. Приклад застосування

Давайте простежимо за станом рангів на невеликому прикладі з 8 елементами.

- Початковий стан:
  - `parent` = {1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8}
  - `rank` = {1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0, 8:0}
- `Union(1, 2)`:
  - `rank[1]==0`, `rank[2]==0`. Ранги однакові.
  - Приєднуємо 2 до 1: `parent[2]=1`.
  - Збільшуємо ранг 1: `rank[1]=1`.
- `Union(3, 4)`:
  - Аналогічно, `parent[4]=3`, `rank[3]=1`.
- `Union(5, 6)`:

- Аналогічно, `parent[6]=5`, `rank[5]=1`.

– `Union(7, 8)`:

- Аналогічно, `parent[8]=7`, `rank[7]=1`.
- Поточний стан: 4 дерева, кожне рангу 1.

– `Union(1, 3)`:

- Корінь для 1 — це 1, корінь для 3 — це 3.
- `rank[1]==1`, `rank[3]==1`. Ранги однакові.
- Приєднуємо 3 до 1: `parent[3]=1`.
- Збільшуємо ранг 1: `rank[1]=2`.
- Тепер у нас є одне велике дерево {1,2,3,4} з коренем 1 і рангом 2.

– `Union(5, 7)`:

- Корінь для 5 — це 5, корінь для 7 — це 7.
- `rank[5]==1`, `rank[7]==1`. Ранги однакові.
- Приєднуємо 7 до 5: `parent[7]=5`.
- Збільшуємо ранг 5: `rank[5]=2`.
- Тепер у нас є два великі дерева, обоє рангу 2.

– `Union(1, 5)`:

- Корінь для 1 — це 1, корінь для 5 — це 5.
- `rank[1]==2`, `rank[5]==2`. Ранги знову однакові.
- Приєднуємо 5 до 1: `parent[5]=1`.
- Збільшуємо ранг 1: `rank[1]=3`.

В результаті ми отримали одне дерево, що об'єднує всі 8 елементів. Його ранг дорівнює 3, що відповідає $\log_2 8$. Висота дерева ніколи не перевищувала логарифм від кількості вершин.

Переваги та недоліки

Переваги

– Значне покращення продуктивності: Гарантує логарифмічний час $O(\log N)$ для операцій `Find` та `Union`, що є величезним стрибком у порівнянні з лінійним часом $O(N)$ наївної реалізації.

– Простота реалізації: Додавання оптимізації вимагає лише одного додаткового масиву та кількох рядків коду в операції `Union`.

– Надійність: Запобігає деградації структури даних у найгіршому випадку.

Недоліки

– Не є остаточною оптимізацією: Хоча $O(\log N)$ — це добре, продуктивність можна зробити ще кращою за допомогою стиснення шляху. Сама по собі ця оптимізація залишає операцію `Find` “пасивною”.

– Додаткові витрати пам’яті: Потребує додаткового масиву для зберігання рангів або розмірів, що, втім, є незначним недоліком.

Контрольні запитання

1. Опишіть сценарій, за якого наївна реалізація DSU деградує до зв’язного списку.
2. У чому полягає основна ідея евристики “об’єднання за рангом”?
3. В якому єдиному випадку ранг дерева збільшується при операції `Union`?
4. Поясніть, чому висота дерева з рангом k не може бути більшою за k.
5. Чому максимальний ранг дерева з N елементів не може перевищувати $\log_2 N$?
6. Чим відрізняється евристика “об’єднання за розміром” від “об’єднання за рангом”?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація DSU з оптимізацією “об’єднання за рангом”.

2. Мета

– Зрозуміти та реалізувати одну з ключових оптимізацій DSU.

– Практично дослідити вплив цієї оптимізації на продуктивність.

– Порівняти продуктивність наївної реалізації та реалізації з оптимізацією.

3. Завдання

– Реалізувати клас або модуль для DSU з наївним методом `Union`.

– Реалізувати другу версію DSU, додавши до неї оптимізацію “об’єднання за рангом”.

– Створити тестовий сценарій, який демонструє найгірший випадок для наївної реалізації (послідовне об’єднання, що створює ланцюжок).

– Провести заміри часу виконання великої кількості операцій `Find` для обох реалізацій на цьому тестовому сценарії.

– Проаналізувати та порівняти отримані результати.

4. Опис

Лабораторна робота фокусується на практичній демонстрації важливості оптимізації “об’єднання за рангом”. Студент повинен реалізувати обидві версії DSU (наївну та оптимізовану) і наочно, за допомогою замірів часу, показати, наскільки великою є різниця в продуктивності, особливо у спеціально створених “поганих” випадках.

5. Порядок виконання

– Реалізація наївної DSU: Створіть клас з методами `MakeSet(v)`, `Find(v)` та `Union(u, v)`, що використовує найпростішу логіку об’єднання.

– Реалізація оптимізованої DSU: Скопіюйте попередній клас і модифікуйте його. Додайте масив `rank`. Змініть `MakeSet` для ініціалізації рангів нулем. Повністю перепишіть `Union` згідно з логікою об’єднання за рангом.

– Створення тесту: Напишіть код, що створює N елементів, а потім у циклі виконує `Union(i, i+1)` для i від 1 до $N - 1$. Це створить ланцюжок для наївної версії.

– Заміри часу: Після створення структури, виконайте велику кількість (напр., *M*) операцій `Find` для елементів, що знаходяться далеко від кореня (напр., `Find(1)`). Заміряйте час виконання цього циклу для обох реалізацій.

– Аналіз: Порівняйте отриманий час. Теоретично, різниця має бути величезною. Опишіть свої спостереження у звіті.

6. Результат

Результатом роботи є дві реалізації DSU, програмний код для їх тестування та порівняння, а також звіт з результатами замірів та висновками щодо ефективності оптимізації.

Після виконання лабораторної роботи студент повинен надати:

– Вихідний код програми, що містить:

  – Реалізацію наївної версії DSU.

  – Реалізацію DSU з оптимізацією “об’єднання за рангом”.

  – Код для тестування та порівняння продуктивності.

– Продемонструвати роботу додатку:

  – Запустити тестування і показати результати замірів часу для обох версій.

  – Пояснити, чому виникла така різниця в продуктивності.

– Бути готовим відповісти на питання:

  – Питання з розділу “Контрольні запитання”.

  – Як би змінився ваш код, якби ви реалізовували “об’єднання за розміром” замість рангу?

  – Чому в вашій реалізації `Find` все ще рекурсивний і потенційно повільний? (Підказка: ми ще не розглянули другу оптимізацію).

  – Яка складність операцій вашої оптимізованої версії?

Тести

1. Основна мета оптимізації “об’єднання за рангом” — це:

а) Зменшити кількість елементів у структурі

b) Зменшити висоту дерев, що представляють множини

c) Прискорити операцію `MakeSet`

d) Зменшити використання пам’яті

2. При об’єднанні двох дерев однакового рангу, ранг результуючого дерева:

a) Залишається тим самим

b) Збільшується на одиницю

c) Дорівнює сумі рангів

d) Обчислюється заново

3. При об’єднанні дерева рангу 3 і дерева рангу 5, ранг результуючого дерева буде:

a) 3

b) 5

c) 8

d) 6

4. Використання оптимізації “об’єднання за рангом” (без інших оптимізацій) гарантує часову складність операції `Find`:

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(N \log N)$

5. Який з варіантів є настільки ж ефективною альтернативою об’єднанню за рангом?

a) Об’єднання за індексом

b) Випадкове об’єднання

c) Об’єднання за розміром

d) Об’єднання в зворотньому порядку

Відповіді:

1. - b)

2. - b)

3. - b)

4. - b)

5. - c)

Тести підвищеної складності

1. Якщо не використовувати стиснення шляху, ранг вершини є:

a) Точною висотою її піддерева.

b) Кількістю елементів у її піддереві.

c) Верхньою межею на висоту її піддерева.

d) Завжди парним числом.

2. Чому при об’єднанні дерев різних рангів ранг більшого дерева не змінюється?

a) Тому що висота більшого дерева не змінюється, коли до його кореня приєднують менше дерево.

b) Це спрощення, щоб не робити зайвих обчислень.

c) Тому що ранг може лише збільшуватися, а не змінюватися.

d) Насправді він змінюється, але це ігнорується.

3. Максимально можливий ранг у DSU з 1,000,000 елементів буде приблизно:

a) 1,000,000

b) 1000

c) 20

d) 10

4. Який головний недолік має евристика “об’єднання за розміром” у порівнянні з “об’єднанням за рангом”?

a) Вона повільніша.

b) Вона потребує більше пам’яті.

c) Жодного, вони практично еквівалентні за ефективністю.

d) Вона не працює зі стисненням шляху.

5. Якщо ми маємо DSU, побудовану з оптимізацією за рангом, і виконаємо `Find(v)`, скільки разів максимум буде викликана рекурсивна функція `Find`?

a) Не більше `rank[корінь(v)]` + 1 разів.

b) Точно `rank[корінь(v)]` разів.

c) Завжди 1 раз.

d) Не більше `size[корінь(v)]` разів.

6. Якщо ми маємо 16 елементів і послідовно об’єднуємо їх парами (1,2), (3,4)..., потім результати парами ((1,2), (3,4))..., який буде ранг фінального кореня?

a) 16

b) 8

c) 4

d) 3

7. Чи гарантує оптимізація “об’єднання за рангом”, що структура дерева буде унікальною для заданої послідовності операцій?

a) Так, структура завжди однакова.

b) Ні, при об’єднанні дерев однакового рангу вибір нового кореня може бути довільним.

c) Тільки якщо всі елементи є числами.

d) Тільки якщо не було об’єднань однакових рангів.

8. Яка асимптотична складність побудови DSU на N елементах з M операціями `Union` (з оптимізацією за рангом)?

a) $O(N + M)$

b) $O(N \log M)$

c) $O(N + M \log N)$

d) $O(N \cdot M)$

9. Що станеться, якщо при об'єднанні дерев однакового рангу забути збільшити ранг нового кореня?

a) Нічого страшного, це лише евристика.

b) Алгоритм видасть помилку.

c) Логарифмічна оцінка висоти перестане бути гарантованою, і структура може деградувати.

d) Це призведе до нескінченної рекурсії.

10. У порівнянні з другою оптимізацією (стиснення шляху), об'єднання за рангом:

a) Є менш важливим.

b) Є більш важливим.

c) Важливе настільки ж, оскільки вони ефективні лише разом.

d) Є застарілим і більше не використовується.

Відповіді на тести підвищеної складності

1. - c) Стиснення шляху може зменшити реальну висоту дерева, але ранг залишається незмінним, тому він є лише верхньою межею.

2. - a) Новий корінь (з більшим рангом) просто "всиновлює" корінь меншого дерева. Його власна висота від цього не змінюється, оскільки новий нащадок приєднується на тому ж рівні, що й існуючі.

3. - c) $\log_2(1{,}000{,}000) \approx 19.9$. Максимальний ранг буде 19 або 20.

4. - c) Обидві евристики забезпечують логарифмічну висоту дерев і мають однакову асимптотичну ефективність. Вибір між ними часто є питанням особистих вподобань.

5. - a) Операція `Find` рухається від вершини до її батька, доки не досягне кореня. Довжина цього шляху обмежена висотою дерева, а висота, в свою чергу, обмежена рангом.

6. - c) (1,2)->r=1; (3,4)->r=1; ... (8 пар рангу 1). ((1,2),(3,4))->r=2; ... (4 пари рангу 2). (((1,2),(3,4)),((5,6),(7,8)))->r=3. Фінальний корінь матиме ранг 3.

7. - b) Коли ранги дерев однакові, ми можемо приєднати праве до лівого або ліве до правого. Цей вибір може вплинути на фінальну структуру, хоча висота буде однаковою.

8. - c) N операцій `MakeSet` займають $O(N)$ часу. M операцій `Union` в найгіршому випадку вимагають $O(\log N)$ кожна. Разом $O(N + M\log N)$.

9. - c) Якщо не збільшувати ранг, то твердження про те, що дерево рангу k має щонайменше 2^k вершин, стає хибним. Це порушує доведення логарифмічної висоти, і структура може виродитись.

10. - c) Обидві оптимізації є критично важливими. Об'єднання за рангом запобігає утворенню довгих ланцюжків, а стиснення шляху "сплющує" дерева, які все одно утворилися б. Разом вони дають майже константний час.

Завдання для самостійної роботи

1. Реалізуйте DSU з оптимізацією “об’єднання за розміром” і порівняйте її з “об’єднанням за рангом”. Перевірте, чи є різниця в продуктивності на практиці.
2. Напишіть функцію, яка для заданої DSU візуалізує її структуру у вигляді дерева (наприклад, виводить у текстовому форматі з відступами).
3. Дослідіть, яка максимальна висота дерева реально утворюється при використанні об’єднання за рангом. Порівняйте її зі значенням рангу.
4. Напишіть програму, яка генерує випадкову послідовність операцій `Union` і будує DSU. Проаналізуйте фінальну структуру.

5. Використайте вашу реалізацію DSU для написання простої версії алгоритму Крускала.
6. Реалізуйте функцію, яка для DSU знаходить кількість неперетинних множин.
7. Реалізуйте функцію, яка для заданого елемента знаходить розмір множини, до якої він належить.
8. Подумайте, як можна було б реалізувати операцію `Delete`, яка видаляє елемент з його множини. Чи можливо це зробити ефективно?
9. Створіть тест, який показує, що без оптимізації за рангом, але з другою оптимізацією (стиснення шляху), структура все ще може деградувати.
10. Прочитайте про застосування DSU для генерації лабіринтів і напишіть короткий опис цього методу.

Теми лабораторних робіт

1. Порівняльний аналіз продуктивності наївної DSU, DSU з об’єднанням за рангом та DSU з об’єднанням за розміром.
2. Реалізація алгоритму Крускала з використанням оптимізованої структури DSU.
3. Використання DSU для динамічного знаходження компонент зв’язності у графі.
4. Візуалізатор структури DSU: програма, що графічно відображає дерева, які утворюються в процесі роботи.
5. Застосування DSU для задачі сегментації зображень.
6. Дослідження впливу порядку операцій `Union` на фінальну структуру та висоту дерев.
7. Реалізація DSU для обробки онлайн-запитів (коли операції надходять послідовно і потрібно швидко давати відповідь).
8. Генерація лабіринтів за допомогою DSU.

9. Аналіз соціальної мережі: групування користувачів у спільноти за допомогою DSU.
10. Повна реалізація DSU з обома оптимізаціями (об’єднання за рангом та стиснення шляху).

Оптимізація “Стиснення шляху”

У попередньому розділі ми зробили значний крок у покращенні DSU, застосувавши оптимізацію “об’єднання за рангом”. Ми досягли гарантованої логарифмічної складності $O(\log N)$ для операцій, не дозволяючи нашим деревам вироджуватися в довгі ланцюжки. Це вже дуже добре, але ми можемо досягти ще більшого.

Проблема, що залишається, полягає в тому, що операція `Find` є “пасивною”. Вона проходить шлях від вершини до кореня, отримує потрібну інформацію, але залишає структуру дерева незмінною. Стиснення шляху — це друга геніальна оптимізація, яка перетворює `Find` на активного “оптимізатора”. Ідея проста: якщо ми вже пройшли шлях до кореня, то чому б не використати цю інформацію, щоб зробити майбутні пошуки швидшими?

1. Теоретичні відомості

1.1. Ідея стиснення шляху

Коли ми викликаємо `Find(v)`, ми знаходимо корінь дерева, до якого належить вершина v. Це означає, що всі вершини на шляху від v до кореня належать до однієї і тієї ж множини. Отже, ми можемо безболісно змінити їхні батьківські вказівники так, щоб вони всі вказували безпосередньо на корінь.

Ця операція “сплющує” дерево. Наступного разу, коли ми викличемо `Find` для будь-якої з цих вершин, ми знайдемо корінь за один крок. Це надзвичайно ефективна “самооптимізація” структури даних.

1.2. Реалізація та псевдокод

Стиснення шляху реалізується шляхом невеликої, але дуже важливої зміни в коді операції `Find`. Операції `MakeSet` та `Union` залишаються такими ж, як і в попередньому розділі.

Операції `MakeSet` та `Union`

Для повноти картини наведемо код усієї структури DSU з обома оптимізаціями. `MakeSet` та `Union` не змінюються порівняно з попереднім розділом.

```
function MakeSet(v, parent, rank):
  parent[v] = v
  rank[v] = 0

function Union_by_Rank(u, v, parent, rank):
  root_u = Find(u, parent) // Викликаємо вже оптимізований Find
  root_v = Find(v, parent) // Викликаємо вже оптимізований Find

  if root_u == root_v:
    return

  if rank[root_u] < rank[root_v]:
    parent[root_u] = root_v
  else if rank[root_u] > rank[root_v]:
    parent[root_v] = root_u
  else:
    parent[root_v] = root_u
    rank[root_u] = rank[root_u] + 1
```

Операція `Find` (зі стисненням шляху)

Найелегантніший спосіб реалізувати стиснення шляху — це рекурсивний однокроковий метод.

```
function Find_with_Path_Compression(v, parent):
  // Базовий випадок рекурсії: ми знайшли корінь
  if v == parent[v]:
    return v

  // Рекурсивно шукаємо корінь для батька вершини v
  root = Find_with_Path_Compression(parent[v], parent)

  // МАГІЯ: Коли рекурсія розгортається, ми переприв'язуємо
  // батька вершини v безпосередньо до знайденого кореня.
  parent[v] = root

  return root
```

Коли ми викликаємо `Find(v)`, рекурсія заглиблюється доти, доки не знайде корінь. Після цього корінь повертається вгору по стеку викликів. На

кожному кроці “розгортання” рекурсії рядок `parent[v] = root` переписує батьківський вказівник поточної вершини v, встановлюючи його на кінцевий корінь. Таким чином, весь пройдений шлях “прив’язується” напряму до кореня.

1.3. Сила двох оптимізацій

Дуже важливо розуміти, що максимальна ефективність досягається лише при одночасному використанні обох оптимізацій: об’єднання за рангом/розміром та стиснення шляху.

- Об’єднання за рангом запобігає утворенню високих дерев, зберігаючи їх “широкими”.
- Стиснення шляху робить ці невисокі дерева ще й “пласкими”.

Разом вони створюють одну з найшвидших відомих структур даних.

1.4. Аналіз складності

Аналіз складності DSU з обома оптимізаціями є дуже нетривіальним і виходить за рамки більшості базових курсів. Проте результат є настільки вражаючим, що його необхідно знати.

- Амортизована складність: Час виконання послідовності з M операцій `MakeSet`, `Union` та `Find` на N елементах становить $O(M \cdot \alpha(N))$, де $\alpha(N)$ — обернена функція Аккермана.
- Це означає, що амортизований час виконання однієї операції становить $O(\alpha(N))$.

Що таке обернена функція Аккермана $\alpha(N)$? Це функція, що росте надзвичайно повільно. Настільки повільно, що для будь-яких значень N, які можуть зустрітися в реальному світі (навіть більших за кількість атомів у Всесвіті), $\alpha(N)$ ніколи не перевищить 5.

Тому на практиці амортизовану складність DSU з двома оптимізаціями можна вважати майже константною. Хоча одна окрема операція `Find` все ще може пройти по довгому шляху ($O(\log N)$), вона настільки покращить структуру, що сумарна вартість багатьох операцій буде дуже низькою.

2. Область застосування

DSU з обома оптимізаціями є “золотим стандартом” і використовується в усіх задачах, де потрібна ця структура даних.

– Конкурентне програмування: Це одна з перших структур даних, яку вивчають для розв'язання задач на графи, оскільки вона надзвичайно швидка і відносно проста в реалізації.

– Алгоритм Крускала: Завдяки майже константному часу роботи DSU, загальна складність алгоритму Крускала визначається лише сортуванням ребер ($O(E\log E)$).

– Високопродуктивні системи: У будь-яких системах, що вимагають швидкого об'єднання множин та перевірки належності до них.

3. Приклад застосування

Давайте подивимося на стиснення шляху в дії. Припустимо, в результаті попередніх операцій у нас утворився такий ланцюжок (що можливо, якщо ми не використовували об'єднання за рангом, але для демонстрації це зручно):

– 1 → 2 → 3 → 4 (4 — корінь).

– `parent` = {1:2, 2:3, 3:4, 4:4}.

Виконуємо `Find(1)`:

1. `Find(1)` викликає `Find(parent[1])`, тобто `Find(2)`.
2. `Find(2)` викликає `Find(parent[2])`, тобто `Find(3)`.
3. `Find(3)` викликає `Find(parent[3])`, тобто `Find(4)`.
4. `Find(4)` бачить, що `parent[4] == 4`. Це базовий випадок. Він повертає `4`.
5. Виклик `Find(3)` отримує результат `4`. Він виконує `parent[3] = 4` і повертає `4`.
6. Виклик `Find(2)` отримує результат `4`. Він виконує `parent[2] = 4` і повертає `4`.
7. Виклик `Find(1)` отримує результат `4`. Він виконує `parent[1] = 4` і повертає `4`.

Результат:

– Після одного виклику `Find(1)` структура кардинально змінилася.

– Новий стан `parent`: {1:4, 2:4, 3:4, 4:4}.

– Тепер `Find(1)`, `Find(2)` і `Find(3)` будуть виконуватися за один крок.

Ми виконали одну трохи “довгу” операцію, але натомість отримали значне прискорення для багатьох майбутніх операцій.

Переваги та недоліки

Переваги

– Неймовірна швидкість: Досягається майже константний амортизований час на операцію, що робить DSU однією з найшвидших структур даних.

– Простота коду: Сама зміна в коді функції `Find` є мінімальною і елегантною.

– Самооптимізація: Структура даних автоматично покращується в процесі її використання.

Недоліки

– Складність аналізу: Теоретичне доведення амортизованої складності $O(\alpha(N))$ є дуже складним.

– Порушення структури рангу: Стиснення шляху змінює висоту дерев. Через це значення `rank`, яке ми зберігаємо, перестає бути точною висотою, а залишається лише її верхньою межею. Саме тому ми від самого початку називали його “рангом”, а не “висотою”. Це не впливає на коректність, але є важливим теоретичним нюансом.

Контрольні запитання

1. У чому полягає основна ідея оптимізації “стиснення шляху”?
2. Яка операція DSU модифікується для реалізації цієї оптимізації?
3. Поясніть на прикладі, як рекурсивна реалізація `Find` досягає стиснення шляху.
4. Чому важливо використовувати стиснення шляху разом з об’єднанням за рангом?
5. Що таке “амортизована складність” і як вона застосовується до DSU?
6. Що таке обернена функція Аккермана і чому її можна вважати майже константою на практиці?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація DSU з обома оптимізаціями та аналіз її продуктивності.

2. Мета

- Реалізувати повну, високоефективну версію структури даних DSU.
- Зрозуміти синергетичний ефект від поєднання двох оптимізацій.
- Практично підтвердити майже константний час роботи операцій.

3. Завдання

- Взяти за основу реалізацію DSU з оптимізацією “об’єднання за рангом” з попередньої роботи.
- Модифікувати в ній функцію `Find` для реалізації оптимізації “стиснення шляху”.
- Провести заміри часу виконання великої кількості операцій `Find` та `Union` на великих наборах даних (наприклад, 10^6 елементів та 10^6 операцій).
- Порівняти отримані результати з часом роботи реалізації, що мала лише одну оптимізацію.
- Проаналізувати результати та зробити висновки.

4. Опис

Це фінальна лабораторна робота по DSU. Студент повинен створити “промислову” версію структури даних, що включає обидві стандартні оптимізації. Головна мета — на практиці побачити, наскільки швидкою є фінальна реалізація, і зрозуміти, чому DSU є настільки популярною структурою даних.

5. Порядок виконання

- Модифікація `Find`: Візьміть код з попередньої роботи. Змініть лише функцію `Find` на рекурсивну версію зі стисненням шляху.
- Створення тесту: Напишіть код, що генерує великий набір випадкових операцій. Наприклад, створіть N елементів, а потім у циклі M разів з певною ймовірністю виконуйте `Union` між двома випадковими елементами або `Find` для випадкового елемента.

– Заміри часу: Заміряйте сумарний час виконання цього циклу з *M* операцій для версії DSU лише з об’єднанням за рангом, а потім для фінальної версії з обома оптимізаціями.

– Аналіз: Порівняйте отриманий час. Ви маєте побачити помітне прискорення навіть у порівнянні з уже оптимізованою версією. Поясніть у звіті, чому це відбувається.

6. Результат

Результатом роботи є повна реалізація DSU з обома оптимізаціями, код для тестування продуктивності та звіт з аналізом результатів, що підтверджує високу ефективність фінальної структури.

Після виконання лабораторної роботи студент повинен надати:

– Вихідний код програми, що містить повну реалізацію DSU з об’єднанням за рангом та стисненням шляху.

– Продемонструвати роботу додатку:

  - Запустити тест на великих даних і показати сумарний час виконання.
  - Пояснити на невеликому прикладі, як саме відбувається стиснення шляху у вашій реалізації `Find`.

– Бути готовим відповісти на питання:

  - Питання з розділу “Контрольні запитання”.
  - Чому після додавання стиснення шляху “ранг” перестає бути точною “висотою” дерева?
  - Яка амортизована складність вашої фінальної реалізації?
  - Чи можна використовувати стиснення шляху без об’єднання за рангом? Які можуть бути наслідки?

Тести

1. Оптимізація “стиснення шляху” застосовується під час виконання якої операції?

a) `MakeSet`

b) `Union`

c) `Find`

d) `Delete`

2. Який основний ефект від застосування стиснення шляху?

a) Зменшення кількості множин

b) Зменшення рангу кореня

c) Зменшення висоти дерева шляхом “сплющення”

d) Збільшення кількості елементів

3. Найкраща продуктивність DSU досягається, коли:

a) Використовується лише об’єднання за рангом

b) Використовується лише стиснення шляху

c) Використовуються обидві оптимізації разом

d) Не використовується жодна з оптимізацій

4. Амортизована складність DSU з двома оптимізаціями становить:

a) $O(1)$

b) $O(\log N)$

c) $O(N)$

d) $O(\alpha(N))$

5. Після виконання `Find(v)` зі стисненням шляху, батьком вершини `v` стає:

a) Її попередній батько

b) Вершина, що стояла перед нею на шляху до кореня

c) Корінь множини

d) Сама вершина `v`

Тести підвищеної складності

1. Чому використання стиснення шляху без об’єднання за рангом не є оптимальним?

a) Це призводить до помилок компіляції.

b) Це не дає жодного прискорення.

c) Можна створити ситуацію, коли довгий ланцюжок приєднується до маленького дерева, і стискати доведеться багато разів.

d) Складність стає експоненційною.

2. Обернена функція Аккермана $\alpha(N)$ для $N = 2^{2^{65536}}$ приблизно дорівнює:

a) 1

b) 5

c) 65536

d) 2^{65536}

3. “Амортизований аналіз” означає, що:

a) Кожна операція гарантовано виконується за вказаний час.

b) Середній час операції у довгій послідовності є низьким, хоча окремі операції можуть бути повільними.

c) Аналіз працює лише для випадкових даних.

d) Структура даних використовує додаткову пам’ять для прискорення.

4. Якщо ми реалізуємо стиснення шляху не рекурсивно, а у два проходи (спочатку знайти корінь, потім пройти шлях ще раз, оновлюючи вказівники), як це вплине на складність?

a) Асимптотична складність залишиться тією ж.

b) Це буде значно повільніше.

c) Це буде значно швидше.

d) Це неможливо реалізувати.

5. Після застосування стиснення шляху, чи залишається `rank` верхньою межею на висоту дерева?

a) Так, ранг ніколи не може бути меншим за реальну висоту.

b) Ні, реальна висота може стати більшою за ранг.

c) Ранг завжди дорівнює висоті.

d) Поняття рангу втрачає сенс.

6. Яка з операцій DSU найбільше виграє від стиснення шляху?

a) `MakeSet`

b) `Union` (опосередковано, через виклики `Find`)

c) `Find`

d) b) і c)

7. Чи можна реалізувати операцію `Find` ітеративно (циклом) зі стисненням шляху за один прохід?

a) Ні, це можливо лише рекурсивно або у два проходи.

b) Так, але це вимагає складніших маніпуляцій з вказівниками.

c) Так, це стандартна і найпростіша реалізація.

d) Ні, цикл не може змінити структуру дерева.

8. Скільки разів потрібно викликати `Find` для одного й того ж елемента, щоб він гарантовано вказував напряму на корінь?

a) Один раз.

b) Два рази.

c) $\log N$ разів.

d) Залежить від висоти дерева.

9. Чи впливає стиснення шляху на кількість операцій `Union` в алгоритмі Крускала?

a) Так, значно зменшує.

b) Ні, кількість операцій `Union` залежить лише від структури графа.

c) Так, значно збільшує.

d) Незначно зменшує.

10. Якщо DSU використовується для знаходження компонент зв’язності, то стиснення шляху:

a) Може об’єднати різні компоненти помилково.

b) Не впливає на результат, лише на швидкість.

c) Потрібне для коректної роботи.

d) Уповільнює процес, бо виконує зайві дії.

Відповіді на тести підвищеної складності

1. - c) Без об'єднання за рангом можна послідовно приєднувати довгий ланцюжок до маленького кореня. Хоча стиснення допоможе після першого `Find`, сам цей перший `Find` може бути дуже довгим.

2. - b) Це число є прикладом "вежі степенів", для якої функція Аккермана дає значення 5. Для всіх менших, практичних чисел, значення буде 4 або менше.

3. - b) Це визначення амортизованого аналізу. Він усереднює вартість операцій по довгій послідовності, згладжуючи дорогі, але рідкісні випадки.

4. - a) Рекурсивна версія і двопрохідна виконують ту саму роботу і мають однакову асимптотичну складність. Рекурсивна версія просто елегантніша і коротша в записі.

5. - a) Ранг збільшується лише при злитті дерев однакового рангу. Стиснення шляху може зменшити висоту, зробивши її меншою за ранг, але ніколи не збільшить її понад ранг. Тому ранг залишається коректною верхньою межею.

6. - d) Стиснення шляху безпосередньо прискорює `Find`. `Union` виграє опосередковано, оскільки він викликає `Find` для пошуку коренів.

7. - b) Так, це можливо. Потрібно в циклі спочатку пройти шлях до кореня, зберігаючи вершини у списку, а потім пройти по цьому списку, оновлюючи батьківські вказівники.

8. - b) Перший виклик `Find(v)` стисне шлях від `v` до кореня. Другий виклик `Find(v)` вже гарантовано пройде по стиснутому шляху, тобто за один крок від `v` до її нового батька (кореня).

9. - b) Кількість операцій `Union` в алгоритмі Крускала дорівнює $|V| - 1$ (для зв'язного графа) і не залежить від швидкості операцій DSU, а лише від кількості вершин.

10. - b) Стиснення шляху не змінює належність елементів до множин, воно лише змінює внутрішню структуру представлення цих множин (дерева). Отже, на коректність це не впливає.

Завдання для самостійної роботи

1. Реалізуйте ітеративну (двопрохідну) версію **`Find`** зі стисненням шляху і порівняйте її продуктивність з рекурсивною.
2. Напишіть функцію, яка після кожної операції **`Find`** виводить структуру дерева, щоб наочно бачити ефект стиснення.
3. Створіть тест, що демонструє неефективність використання стиснення шляху без об'єднання за рангом.
4. Дослідіть, як змінюється загальний час виконання M операцій на N елементах при зміні співвідношення **`Find`**/**`Union`** (напр., 90% **`Find`** і 10% **`Union`** проти 10% **`Find`** і 90% **`Union`**).
5. Використайте вашу фінальну реалізацію DSU для вирішення задачі з онлайн-судді (напр., з LeetCode, Codeforces) на компоненти зв'язності.
6. Подумайте, чи можна додати до DSU операцію "скасування" останнього **`Union`** (Undo). Як би ви її реалізували?
7. Напишіть програму, яка будує DSU, а потім обчислює середню довжину шляху до кореня для всіх елементів. Запустіть її до і після серії операцій **`Find`**, щоб побачити ефект стиснення.
8. Прочитайте про застосування DSU в алгоритмі Тарджана для пошуку найближчого спільного предка (LCA) офлайн. Напишіть короткий опис.
9. Реалізуйте DSU, де батьківський вказівник і ранг зберігаються в одній змінній (за допомогою бітових операцій) для економії пам'яті.
10. Дослідіть, як можна було б розпаралелити операції в DSU. Які виникають проблеми?

Теми лабораторних робіт

1. Фінальна реалізація DSU та її застосування для вирішення задачі Labyrinth (пошук шляху в лабіринті).
2. Розробка програми для аналізу перколяції в решітках з використанням DSU.
3. Порівняльний аналіз продуктивності DSU (з обома оптимізаціями) та інших структур даних (напр., на основі списків) для задачі про компоненти зв’язності.
4. Створення інтерактивного візуалізатора DSU, що дозволяє користувачу виконувати операції і бачити зміни в структурі.
5. Використання DSU для реалізації алгоритму для обробки зображень (напр., визначення зв’язних областей).
6. Реалізація DSU зі “скасуванням” операцій (Persistent DSU або DSU with rollback).
7. Застосування DSU для вирішення задач з геометрії (напр., перевірка перетину багатокутників).
8. Аналіз продуктивності DSU на різних моделях випадкових графів (Ердеша-Рені, Барабаші-Альберт).
9. Розробка багатопотокової версії DSU (з блокуваннями для безпечного доступу).
10. Використання DSU як частини складнішого алгоритму (напр., узагальнений алгоритм Крускала для дерева Штейнера).

Частина V: Спеціалізовані алгоритми та структури

Розділ 9: Сортування та порядкові статистики

9.1. Прості сортування

9.1.1 Бульбашкове сортування

Теоретичні відомості

Бульбашкове сортування — це один з найпростіших для розуміння алгоритмів сортування, який функціонує на основі порівнянь. Його назва походить від аналогії з бульбашками повітря у воді, які піднімаються на поверхню. В алгоритмі, на кожній ітерації (проході), найбільший елемент з невідсортованої частини масиву “спливає” на свою правильну позицію в кінці цієї частини.

Процес полягає в багаторазових проходах по масиву. Під час кожного проходу алгоритм послідовно порівнює сусідні елементи і міняє їх місцями, якщо вони знаходяться в неправильному порядку (наприклад, для сортування за зростанням, якщо попередній елемент більший за наступний).

Основний принцип роботи:

1. Алгоритм починає прохід з початку масиву.

2. Він порівнює перший та другий елементи. Якщо перший більший за другий, вони міняються місцями.

3. Далі він порівнює другий та третій елементи, і так далі, до кінця масиву.

4. Після першого повного проходу найбільший елемент гарантовано опиниться на останній позиції в масиві.

5. На другому проході алгоритм повторює ті ж дії, але вже до передостаннього елемента, оскільки останній вже на своєму місці.

6. Цей процес повторюється `n-1` разів, де `n` — кількість елементів у масиві. Після кожного проходу розмір невідсортованої частини масиву зменшується на один елемент.

Псевдокод базового алгоритму

```
function BubbleSort(array A)
  n = length(A)
  // Зовнішній цикл для кількості проходів
  for i from 0 to n-2
    // Внутрішній цикл для порівняння сусідніх елементів
    for j from 0 to n-i-2
      // Порівняння пари сусідніх елементів
      if A[j] > A[j+1]
        // Обмін елементів, якщо вони в неправильному порядку
        swap(A[j], A[j+1])
      end if
    end for
```

```
  end for
end function
```
- `A`: Масив, який потрібно відсортувати.

- `n`: Кількість елементів у масиві `A`.

- `i`: Лічильник зовнішнього циклу, який відповідає за номер проходу. Після `i`-го проходу `i` найбільших елементів вже знаходяться на своїх місцях в кінці масиву. Тому внутрішній цикл можна скоротити.

- `j`: Лічильник внутрішнього циклу, який проходить по невідсортованій частині масиву.

Оптимізована версія

Основний недолік базової версії полягає в тому, що вона виконує всі `n-1` проходів, навіть якщо масив уже відсортовано. Це можна оптимізувати. Якщо під час повного проходу внутрішнього циклу не відбулося жодного обміну, це означає, що масив вже повністю відсортований, і алгоритм можна завершити достроково. Для цього вводиться логічна змінна (прапорець).

Псевдокод оптимізованого алгоритму

```
function OptimizedBubbleSort(array A)
  n = length(A)
  // Зовнішній цикл для кількості проходів
  for i from 0 to n-2
    swapped = false // Прапорець для відстеження обмінів
    // Внутрішній цикл для порівняння сусідніх елементів
    for j from 0 to n-i-2
      if A[j] > A[j+1]
        swap(A[j], A[j+1])
        swapped = true // Обмін відбувся
      end if
    end for
    // Якщо за весь прохід не було обмінів, масив відсортований
    if swapped == false
      break // Дострокове завершення
    end if
  end for
end function
```
Приклад роботи (покроково)

Розглянемо сортування масиву `A = [5, 1, 4, 2, 8]` за допомогою оптимізованого алгоритму.

- **Прохід 1:**

- `swapped` встановлено в `false`.

- `(5, 1)`: `5 > 1`. Міняємо. Масив: `[1, 5, 4, 2, 8]`. `swapped = true`.

- `(5, 4)`: `5 > 4`. Міняємо. Масив: `[1, 4, 5, 2, 8]`. `swapped = true`.

- `(5, 2)`: `5 > 2`. Міняємо. Масив: `[1, 4, 2, 5, 8]`. `swapped = true`.

- `(5, 8)`: `5 < 8`. Нічого не робимо.

- Кінець проходу. Найбільший елемент `8` тепер на своєму місці. Оскільки `swapped` є `true`, продовжуємо.

- **Прохід 2:**

- `swapped` встановлено в `false`.

- `(1, 4)`: `1 < 4`. Нічого не робимо.

- `(4, 2)`: `4 > 2`. Міняємо. Масив: `[1, 2, 4, 5, 8]`. `swapped = true`.

- `(4, 5)`: `4 < 5`. Нічого не робимо.

- Кінець проходу. Елемент `5` тепер на своєму місці. Оскільки `swapped` є `true`, продовжуємо.

- **Прохід 3:**

- `swapped` встановлено в `false`.

- `(1, 2)`: `1 < 2`. Нічого не робимо.

- `(2, 4)`: `2 < 4`. Нічого не робимо.

- Кінець проходу. `swapped` залишається `false`. Алгоритм бачить, що обмінів не було, і завершує роботу.

Аналіз складності

- **Часова складність:**

- **Найгірший випадок: $O(n^2)$**. Це відбувається, коли масив відсортований у зворотному порядку. Алгоритму доведеться виконати `n-1` проходів, і на кожному проході внутрішній цикл буде виконувати максимальну кількість порівнянь та обмінів.

- **Середній випадок: $O(n^2)$**. Для випадково розташованих елементів кількість порівнянь та обмінів також буде пропорційною квадрату кількості елементів.

- **Найкращий випадок: O(n)**. Це стосується **оптимізованої версії** алгоритму. Якщо масив вже відсортований, алгоритм зробить один прохід, не виконає жодного обміну і завершить роботу.

- **Просторова складність: O(1)**. Бульбашкове сортування є “in-place” алгоритмом, тобто йому не потрібна додаткова пам’ять, пропорційна розміру вхідних даних. Єдиний додатковий простір — це тимчасова змінна для операції обміну (`swap`).

Властивості алгоритму

- **Стійкість:** Алгоритм є **стійким**. Це означає, що відносний порядок елементів з однаковими значеннями не змінюється після сортування. Це відбувається тому, що обмін відбувається лише тоді, коли `A[j] > A[j+1]`. Якщо елементи рівні, обміну не буде.

Область застосування

Через високу часову складність $O(n^2)$, бульбашкове сортування є неефективним для великих масивів даних і рідко використовується в реальних промислових системах. Однак воно має свої ніші:

- **Освітні цілі:** Це один з перших алгоритмів сортування, який вивчають студенти, оскільки його логіка дуже проста для розуміння та візуалізації.

- **Малі набори даних:** Для дуже невеликих масивів (наприклад, менше 20 елементів) різниця у швидкодії між $O(n^2)$ та більш ефективними алгоритмами, як-от $O(n \log n)$, може бути незначною, а простота реалізації бульбашкового сортування стає перевагою.

- **Перевірка майже відсортованих масивів:** Оптимізована версія може бути корисною для перевірки, чи є масив вже відсортованим, або для сортування масивів, що містять лише кілька елементів не на своїх місцях.

Приклади застосування

Уявімо, що у вас є невелика таблиця лідерів у простій грі, де зберігається до 10 гравців. Коли новий гравець додає свій результат, список потрібно пересортувати. Оскільки n дуже мале, використання простого бульбашкового сортування для оновлення списку є цілком прийнятним і легким для реалізації.

Інший приклад — етап попередньої обробки даних, де потрібно відсортувати невеликі блоки даних, перш ніж передати їх для подальшої, більш складної обробки.

Переваги та недоліки

Переваги:

- **Простота реалізації:** Алгоритм легко написати та зрозуміти.

- **Мінімальне використання пам'яті:** Просторова складність `O(1)`, що робить його ефективним з точки зору пам'яті.

- **Стійкість:** Зберігає відносний порядок однакових елементів.

- **Ефективність на відсортованих даних:** Оптимізована версія має найкращий час `O(n)`.

Недоліки:

- **Низька ефективність:** Часова складність $O(n^2)$ у середньому та найгіршому випадках робить його вкрай повільним для великих наборів даних.

- **Велика кількість обмінів:** Навіть у порівнянні з іншими $O(n^2)$ алгоритмами, як-от сортування вибором, бульбашкове сортування може виконувати значно більше операцій обміну.

Контрольні запитання

1. Поясніть, чому алгоритм називається "бульбашковим". Який елемент "спливає" на кожній ітерації?

2. У чому полягає основна оптимізація бульбашкового сортування і як вона впливає на часову складність у найкращому випадку?

3. Наведіть приклад масиву з 5 елементів, для якого бульбашкове сортування виконає максимальну кількість порівнянь та обмінів.

4. Чи є бульбашкове сортування стійким? Поясніть свою відповідь на прикладі.

5. Порівняйте бульбашкове сортування з сортуванням вибором (Selection Sort) за кількістю операцій обміну в найгіршому випадку.

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз продуктивності алгоритму бульбашкового сортування.

2. Мета:

- Навчитися реалізовувати базову та оптимізовану версії алгоритму бульбашкового сортування.

- Дослідити часову складність алгоритму на практиці, тестуючи його на різних типах даних (випадкові, відсортовані, відсортовані у зворотному порядку).

- Порівняти продуктивність обох версій алгоритму.

3. Завдання:

- Реалізувати функцію бульбашкового сортування у базовому варіанті.

- Реалізувати функцію бульбашкового сортування в оптимізованому варіанті з прапорцем для дострокового виходу.

- Створити допоміжні функції для генерації масивів: випадкового, відсортованого та відсортованого у зворотному порядку.

- Написати програму, яка вимірює час виконання обох версій алгоритму на масивах різного розміру та типу.

4. Опис: Лабораторна робота полягає у створенні програми, яка реалізує два варіанти бульбашкового сортування. Студент повинен провести експериментальне дослідження, щоб підтвердити теоретичні оцінки складності. Результати вимірювань часу виконання слід проаналізувати та зробити висновки щодо ефективності алгоритму та його оптимізації.

5. Порядок виконання:

- Створити проект у вибраному середовищі розробки.

- Написати функцію, що реалізує базовий алгоритм `BubbleSort`.

- Написати функцію, що реалізує оптимізований алгоритм `OptimizedBubbleSort`.

- Написати функцію для генерації масивів заданого розміру та типу.

- У головній функції програми послідовно:

- Згенерувати масиви різного розміру (наприклад, 100, 1000, 5000, 10000 елементів).

- Для кожного розміру створити три типи масивів: випадковий, відсортований, зворотний.

- Для кожного масиву запустити обидві версії сортування, вимірюючи час виконання (наприклад, за допомогою системного таймера).

- Вивести результати вимірювань у зручному вигляді.

- Підготувати звіт, що містить аналіз отриманих результатів.

6. Результат:

- Програма, що коректно реалізує та тестує алгоритми.

- Звіт з таблицями або графіками, що демонструють залежність часу виконання від розміру та типу вхідних даних для обох версій алгоритму.

- Висновки, що підтверджують теоретичні знання про переваги оптимізованої версії та загальну неефективність алгоритму на великих даних.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:

    - Функція базового бульбашкового сортування.

    - Функція оптимізованого бульбашкового сортування.

    - Функції для генерації тестових даних.

- Продемонструвати роботу додатку:

    - Запуск сортування для невеликого масиву з виведенням проміжних кроків.

    - Запуск вимірювань часу для великих масивів та порівняння результатів.

- Бути готовим відповісти на питання:

- Яка часова та просторова складність алгоритму?

- У чому різниця між базовою та оптимізованою версіями?

- Який тип вхідних даних є найкращим, а який — найгіршим для алгоритму?

- Чому цей алгоритм не використовується для великих обсягів даних на практиці?

Тести

1. Яка часова складність у найгіршому випадку для бульбашкового сортування?

1) O(n log n)

2) O(n)

3) $O(n^2)$

4) O(1)

2. Що станеться, якщо застосувати оптимізоване бульбашкове сортування до вже відсортованого масиву?

1) Алгоритм виконає n-1 проходів без обмінів.

2) Алгоритм виконає один прохід і завершиться.

3) Алгоритм виконає приблизно n/2 проходів.

4) Алгоритм матиме складність $O(n^2)$.

3. Яка основна перевага бульбашкового сортування?

1) Висока швидкість на великих масивах.

2) Простота реалізації та розуміння.

3) Ефективне використання кешу процесора.

4) Найкраща середня часова складність серед простих сортувань.

4. Скільки обмінів виконає бульбашкове сортування для масиву `[3, 2, 1]`?

1) 1

2) 2

3) 3

4) 4

5. Бульбашкове сортування є стійким. Що це означає?

1) Алгоритм завжди працює з однаковою швидкістю.

2) Алгоритм не використовує додаткової пам’яті.

3) Алгоритм не змінює відносний порядок елементів з однаковими значеннями.

4) Алгоритм може сортувати лише числа.

Відповіді:

1. c)
2. b)
3. b)
4. c)
5. c)

Тести підвищеної складності

1. Розгляньте масив `A = [6, 2, 7, 3, 9, 4]`. Яким буде стан масиву після завершення другого повного проходу (зовнішнього циклу `i=1`) бульбашкового сортування?

1) `[2, 3, 6, 7, 4, 9]`

2) `[2, 6, 3, 7, 4, 9]`

3) `[2, 3, 4, 6, 7, 9]`

4) `[2, 6, 7, 3, 4, 9]`

2. Яку максимальну кількість обмінів може виконати бульбашкове сортування для масиву з `n` елементів?

1) `n - 1`

2) `n * (n - 1) / 2`

3) n^2

4) `n`

3. В оптимізованій версії бульбашкового сортування прапорець `swapped` допомагає:

1) Зменшити кількість порівнянь.

2) Зменшити кількість обмінів.

3) Достроково завершити роботу на частково або повністю відсортованих масивах.

4) Гарантувати стійкість алгоритму.

4. Для якого з наведених масивів оптимізоване бульбашкове сортування буде найменш ефективним (виконає найбільше роботи)?

1) `[1, 2, 3, 4, 5]`

2) `[2, 1, 3, 4, 5]`

3) `[5, 1, 2, 3, 4]`

4) `[5, 4, 3, 2, 1]`

5. Розглянемо Cocktail Shaker Sort (двонаправлене бульбашкове сортування), яке на кожній ітерації проходить масив спочатку зліва направо (переміщуючи найбільший елемент в кінець), а потім справа наліво (переміщуючи найменший елемент на початок). Як це вплине на складність у найгіршому випадку?

1) Покращить до O(n log n).

2) Погіршить до $O(n^3)$.

3) Не змінить асимптотичної складності, вона залишиться $O(n^2)$.

4) Покращить до O(n).

6. Скільки порівнянь (не обмінів) виконає оптимізоване бульбашкове сортування на масиві `[1, 2, 4, 3, 5]`?

1) 4

2) 7

3) 10

4) 6

7. Яка зміна в коді бульбашкового сортування може зробити його нестійким?

1) Заміна `>` на `>=` в умові порівняння `if A[j] > A[j+1]`.

2) Сортування у зворотному порядку (`<` замість `>`).

3) Використання рекурсії замість циклів.

4) Жодна з перелічених змін не вплине на стійкість.

8. Якщо бульбашкове сортування працює на зв'язному списку замість масиву, як це вплине на його складність?

1) Просторова складність стане O(n).

2) Часова складність покращиться до O(n log n).

3) Часова та просторова складності залишаться $O(n^2)$ та O(1) відповідно.

4) Алгоритм неможливо реалізувати для зв'язного списку.

9. Яку кількість проходів (ітерацій зовнішнього циклу) виконає оптимізоване бульбашкове сортування для масиву `[3, 1, 2, 4, 5]`?

1) 1

2) 2

3) 3

4) 4

10. Бульбашкове сортування іноді називають "Sinking Sort" (сортування потопанням). Який елемент "тоне" в цьому випадку?

1) Найбільший елемент "тоне" на початок.

2) Найменший елемент "тоне" в кінець.

3) Найменший елемент "тоне" на свою позицію на початку невідсортованої частини.

4) Це просто інша назва без зміни логіки.

Відповіді:

1. 1)

Прохід 1: `[6,2,7,3,9,4]` -> `[2,6,7,3,9,4]` -> `[2,6,3,7,9,4]` -> `[2,6,3,7,4,9]`.

Прохід 2: `(2,6)`->ok; `(6,3)`->swap `[2,3,6,7,4,9]`; `(6,7)`->ok; `(7,4)`->swap `[2,3,6,4,7,9]`.

2. 2) Це відбувається у найгіршому випадку (зворотно відсортований масив), де кожен елемент потрібно "протягнути" через весь масив.

3. 3) Достроково завершити роботу на частково або повністю відсортованих масивах. Прапорець перевіряє, чи була виконана хоча б одна операція обміну за прохід.

4. 4) Зворотно відсортований масив є найгіршим випадком, що вимагає максимальної кількості порівнянь та обмінів.

5. 3) Не змінить асимптотичної складності, вона залишиться $O(n^2)$. Хоча це може покращити продуктивність на деяких наборах даних (як-от "черепахи" - малі елементи в кінці масиву), асимптотика в найгіршому випадку залишається тією ж.

6. 2)

Прохід 1: `(1,2)` `(2,4)` `(4,3)->swap` `(4,5)`. 4 порівняння. Масив: `[1,2,3,4,5]`.

Прохід 2: `(1,2)` `(2,3)` `(3,4)`. 3 порівняння. `swapped=false`, кінець. Всього 4+3=7.

7. 1) Заміна `>` на `>=` в умові порівняння `if A[j] > A[j+1]`. Це призведе до обміну однакових елементів, що порушить їх початковий відносний порядок.

8. 3) Часова та просторова складності залишаться $O(n^2)$ та $O(1)$ відповідно. Доступ до сусіднього елемента в зв'язному списку є операцією $O(1)$, тому асимптотика не змінюється.

9. 3)

Прохід 1: `[1,2,3,4,5]`.

Прохід 2: `[1,2,3,4,5]` (один обмін `3` і `1`).

Прохід 3: без обмінів, вихід.

10. 3) Найменший елемент “тоне” на свою позицію на початку невідсортованої частини. Це інша перспектива: якщо великі елементи “спливають”, то малі, відповідно, “тонуть”.

Завдання для самостійної роботи

1. Реалізуйте рекурсивну версію бульбашкового сортування.

2. Напишіть програму, яка візуалізує процес бульбашкового сортування, наприклад, за допомогою стовпчикової діаграми, що оновлюється після кожного обміну.

3. Модифікуйте алгоритм для сортування масиву об’єктів (наприклад, “Студент” з полями “Ім’я” та “Оцінка”) за одним із полів.

4. Дослідіть та реалізуйте “Cocktail Shaker Sort” (двонаправлене бульбашкове сортування) та порівняйте його продуктивність з класичною оптимізованою версією на різних наборах даних.

5. Напишіть функцію, яка підраховує точну кількість порівнянь та обмінів для бульбашкового сортування на заданому масиві.

6. Адаптуйте алгоритм для сортування двовимірного масиву (матриці) за елементами певного стовпця.

7. Дослідіть, як попереднє часткове сортування впливає на швидкість роботи оптимізованого бульбашкового сортування. Створіть “майже відсортований” масив (наприклад, відсортований масив з 5% випадково переставлених елементів) і виміряйте час.

8. Реалізуйте бульбашкове сортування для двобічно зв’язного списку. Проаналізуйте складнощі.

9. Порівняйте практичну продуктивність бульбашкового сортування з сортуванням вставками (Insertion Sort) на малих та майже відсортованих масивах.

10. Придумайте та опишіть сценарій, де використання бульбашкового сортування може бути виправданим у сучасній програмній системі.

Теми лабораторних робіт

1. Створення інтерактивного веб-візуалізатора для алгоритмів простого сортування (Бульбашкове, Вибором, Вставками).

2. Порівняльний аналіз продуктивності простих сортувань ($O(n^2)$) на різних архітектурах процесорів.

3. Розробка гібридного алгоритму: використання `QuickSort` для великих масивів з переходом на `BubbleSort` для малих підмасивів (менше 16 елементів).

4. Аналіз стійкості алгоритмів сортування: реалізація та тестування `BubbleSort` та `QuickSort` на даних з однаковими ключами.

5. Оптимізація бульбашкового сортування для даних з відомим розподілом (наприклад, коли відомо, що жоден елемент не знаходиться далі ніж на `k` позицій від свого фінального місця).

6. Застосування бульбашкового сортування для сортування даних, що надходять у реальному часі в невеликому буфері.

7. Дослідження впливу операцій обміну на продуктивність: порівняння `BubbleSort` та `SelectionSort` за кількістю записів у пам'ять.

8. Реалізація бібліотеки сортування для користувацьких структур даних з можливістю передачі власної функції порівняння (компаратора).

9. Створення системи тестування для алгоритмів сортування, що автоматично генерує звіти про продуктивність на стандартних наборах даних.

10. Адаптація бульбашкового сортування для паралельного виконання на багатоядерних системах (наприклад, через розбиття масиву та подальше злиття).

9.1.2 Сортування вибором

Теоретичні відомості

Сортування вибором — це простий, інтуїтивно зрозумілий алгоритм сортування, що працює "на місці". Його основна ідея полягає в тому, щоб розділити масив на дві частини: відсортовану, яка поступово зростає зліва направо, та невідсортовану, яка відповідно зменшується.

Уявіть, що ви тримаєте в руках набір карток і хочете впорядкувати їх за зростанням. Ви переглядаєте всі картки, знаходите найменшу, виймаєте її і кладете першою. Потім ви знову переглядаєте решту карток, знаходите серед них найменшу і кладете її другою. Цей процес повторюється, доки всі картки не будуть впорядковані. Саме за таким принципом і працює сортування вибором.

Основний принцип роботи:

Алгоритм виконує `n-1` проходів (ітерацій) для сортування масиву з `n` елементів.

1. **Прохід 1:** Знайти найменший елемент у всьому масиві (від індексу 0 до `n-1`). Поміняти його місцями з елементом, що стоїть на першій позиції (індекс 0). Тепер перший елемент є відсортованим.

2. **Прохід 2:** Знайти найменший елемент у решті масиву (від індексу 1 до `n-1`). Поміняти його місцями з елементом на другій позиції (індекс 1). Тепер перші два елементи є відсортованими.

3. **Прохід i:** Знайти найменший елемент у підмасиві від індексу `i-1` до `n-1`. Поміняти його місцями з елементом на позиції `i-1`.

4. Процес триває доти, доки весь масив, крім останнього елемента (який автоматично стане на своє місце), не буде відсортовано.

Псевдокод алгоритму

```
function SelectionSort(array A)
  n = length(A)
  // Зовнішній цикл, який проходить по масиву. i - межа між відсортов
аною і невідсортованою частинами
  for i from 0 to n-2
    // Припускаємо, що перший елемент невідсортованої частини є мінім
альним
    min_idx = i

    // Внутрішній цикл для пошуку індексу мінімального елемента в нев
ідсортованій частині
    for j from i+1 to n-1
      // Якщо знайдено елемент, менший за поточний мінімум
      if A[j] < A[min_idx]
        // Оновлюємо індекс мінімального елемента
        min_idx = j
      end if
    end for

    // Після завершення внутрішнього циклу, min_idx вказує на найменш
ий елемент.
    // Міняємо його місцями з першим елементом невідсортованої частин
и (A[i])
    if min_idx != i
        swap(A[i], A[min_idx])
    end if
```

```
  end for
end function
```
- `A`: Масив, який сортується.

- `n`: Кількість елементів у масиві.

- `i`: Лічильник зовнішнього циклу. Елементи `A[0...i-1]` вже відсортовані.

- `min_idx`: Змінна для зберігання індексу найменшого елемента, знайденого в невідсортованій частині масиву `A[i...n-1]`.

- `j`: Лічильник внутрішнього циклу для сканування невідсортованої частини.

- `swap(A[i], A[min_idx])`: Функція, що міняє місцями два елементи масиву.

Приклад роботи

Розглянемо сортування масиву `A = [64, 25, 12, 22, 11]`.

- **Прохід 1 (i=0):**

- Невідсортована частина: `[64, 25, 12, 22, 11]`.

- Шукаємо мінімум. `min_idx` спочатку `0` (значення 64).

- Порівнюємо з `25` (стає `min_idx=1`), потім з `12` (стає `min_idx=2`), потім з `22`, потім з `11` (стає `min_idx=4`).

- Мінімальний елемент `11` (на індексі 4).

- Міняємо `A[0]` та `A[4]`.

- Масив: `[**11**, 25, 12, 22, 64]`.

Відсортована частина: `[11]`.

- **Прохід 2 (i=1):**

- Невідсортована частина: `[25, 12, 22, 64]`.

- Шукаємо мінімум. `min_idx` спочатку `1` (значення 25).

- Порівнюємо з `12` (стає `min_idx=2`), потім з `22`, потім з `64`.

- Мінімальний елемент `12` (на індексі 2).

- Міняємо `A[1]` та `A[2]`.

- Масив: `[11, **12**, 25, 22, 64]`.

Відсортована частина: `[11, 12]`.

- **Прохід 3 (i=2):**

- Невідсортована частина: `[25, 22, 64]`.

- Шукаємо мінімум. `min_idx` спочатку `2` (значення 25).

- Порівнюємо з `22` (стає `min_idx=3`), потім з `64`.

- Мінімальний елемент `22` (на індексі 3).

- Міняємо `A[2]` та `A[3]`.

- Масив: `[11, 12, **22**, 25, 64]`.

Відсортована частина: `[11, 12, 22]`.

- **Прохід 4 (i=3):**

- Невідсортована частина: `[25, 64]`.

- Шукаємо мінімум. `min_idx` спочатку `3` (значення 25).

- Порівнюємо з `64`. `min_idx` не змінюється.

- Мінімальний елемент `25` (на індексі 3). `min_idx` дорівнює `i`, обмін не потрібен.

- Масив: `[11, 12, 22, **25**, 64]`.

Відсортована частина: `[11, 12, 22, 25]`.

Останній елемент `64` автоматично опиняється на своєму місці. Масив відсортовано.

Аналіз складності

- **Часова складність: O(n²)**.

- Кількість порівнянь не залежить від початкового стану масиву. На першій ітерації виконується `(n-1)` порівнянь, на другій — `(n-2)`, і так далі.

- Загальна кількість порівнянь: $C(n) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$.

- Це рівняння є квадратичною функцією від `n`, тому асимптотична складність становить $O(n^2)$. Вона однакова для **найкращого, середнього та найгіршого** випадків.

- **Просторова складність: O(1)**. Алгоритм є “in-place”, оскільки для його роботи потрібна лише фіксована кількість додаткових змінних (`i`, `j`, `min_idx` та тимчасова змінна для обміну), яка не залежить від розміру масиву.

Властивості алгоритму

- **Кількість обмінів:** На відміну від бульбашкового сортування, яке може виконувати до $O(n^2)$ обмінів, сортування вибором робить не більше `n-1` обмінів. Це його ключова перевага.

- **Стійкість:** Алгоритм є **нестійким**. Стійкий алгоритм зберігає відносний порядок елементів з однаковими значеннями. Сортування вибором може його порушити.

- *Приклад нестійкості:* Розглянемо масив `[5a, 5b, 2]`, де `5a` і `5b` — однакові за значенням, але різні елементи.

- На першому проході мінімальним елементом буде `2`. Алгоритм поміняє його з `5a`.

- Масив стане: `[2, 5b, 5a]`. Відносний порядок `5a` і `5b` змінився.

Область застосування

Як і бульбашкове сортування, сортування вибором має квадратичну складність, що робить його неефективним для великих масивів. Однак, завдяки мінімальній кількості обмінів, воно знаходить застосування в специфічних умовах:

- **Освітні цілі:** Простота алгоритму робить його чудовим кандидатом для вивчення базових концепцій сортування.

- **Ситуації з дорогою операцією запису:** Коли операція обміну елементів (запису в пам'ять) є значно дорожчою за операцію порівняння (читання). Наприклад, при сортуванні даних на флеш-пам'яті, яка має обмежену кількість циклів перезапису.

- **Малі набори даних:** Для невеликих `n` різниця в продуктивності між $O(n^2)$ та $O(n\log n)$ є несуттєвою.

Приклади застосування

Уявіть систему моніторингу, яка збирає невеликі пакети даних (наприклад, по 100 значень) на пристрій з флеш-пам'яттю. Щоб подовжити термін служби пам'яті, важливо мінімізувати кількість операцій запису. У цьому випадку сортування вибором буде кращим вибором, ніж бульбашкове сортування, оскільки воно виконає не більше 99 операцій запису (обмінів), тоді як бульбашкове могло б виконати тисячі.

Переваги та недоліки

Переваги:

- **Простота реалізації:** Легко зрозуміти та імплементувати.

- **Мінімальна кількість обмінів:** Виконує не більше `n-1` обмінів, що є оптимальним для алгоритмів, що працюють на місці.

- **Ефективність по пам'яті:** Просторова складність `O(1)`.

- **Передбачуваність:** Час роботи не залежить від початкового порядку елементів, що спрощує аналіз.

Недоліки:

- **Низька ефективність:** Часова складність $O(n^2)$ в усіх випадках робить його дуже повільним для великих наборів даних.

- **Нестійкість:** Не зберігає відносний порядок однакових елементів.

- **Відсутність адаптивності:** Не прискорюється, якщо масив вже відсортований або майже відсортований, на відміну від оптимізованого бульбашкового сортування чи сортування вставками.

Контрольні запитання

1. Поясніть, чому сортування вибором завжди виконує однакову кількість порівнянь, незалежно від того, чи є масив відсортованим, чи ні.

2. Порівняйте сортування вибором та бульбашкове сортування за максимальною кількістю операцій обміну. В якій ситуації це може бути критично важливим?

3. Наведіть приклад масиву з 4 елементів, включаючи два однакових, і покажіть крок за кроком, як сортування вибором порушує їхній відносний порядок.

4. Чи можна зробити сортування вибором стійким? Якщо так, то як? Якою буде ціна такої модифікації?

5. Як зміниться алгоритм, якщо потрібно відсортувати масив за спаданням, а не за зростанням?

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз алгоритму сортування вибором.

2. Мета:

- Практична реалізація алгоритму сортування вибором.

- Аналіз його часової складності на різних типах даних.

- Порівняльний аналіз кількості операцій обміну та порівнянь з бульбашковим сортуванням.

3. Завдання:

- Реалізувати функцію, що виконує сортування вибором.

- Додати до реалізації лічильники для підрахунку точної кількості порівнянь та обмінів.

- Створити функції для генерації масивів: випадкового, відсортованого та відсортованого у зворотному порядку.

- Провести експерименти для вимірювання часу виконання та кількості операцій на масивах різного розміру та типу.

4. Опис: У ході лабораторної роботи студент повинен реалізувати сортування вибором і доповнити його інструментами для аналізу. Головна мета — не просто отримати відсортований масив, а й зібрати дані про внутрішню роботу алгоритму (кількість ключових операцій) та порівняти їх з теоретичними оцінками та з іншим простим алгоритмом сортування.

5. Порядок виконання:

- Створити проект та реалізувати функцію `SelectionSort`.

- Всередині функції додати глобальні або передані за посиланням лічильники `comparison_count` та `swap_count`.

- Написати програму, яка:

    - Генерує масиви різного розміру (наприклад, 100, 1000, 5000 елементів).

    - Для кожного розміру створює копії масиву для тестування (випадковий, відсортований, зворотний).

    - Запускає сортування, фіксує час виконання та значення лічильників.

    - Виводить результати у вигляді звіту.

    - Для порівняння, провести аналогічні виміри для реалізації бульбашкового сортування з попередньої роботи.

- Підготувати звіт з аналізом результатів.

6. Результат:

- Програма, що коректно реалізує та аналізує сортування вибором.

- Звіт, що містить порівняльні таблиці/графіки часу виконання, кількості порівнянь та обмінів для сортування вибором та бульбашкового сортування.

- Висновки щодо сильних та слабких сторін сортування вибором.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:

- Функція сортування вибором.

- Механізми підрахунку операцій порівняння та обміну.

- Продемонструвати роботу додатку:

- Сортування невеликого масиву з покроковим виведенням стану та лічильників.

- Порівняльний запуск на великих масивах для демонстрації різниці в кількості обмінів з бульбашковим сортуванням.

- Бути готовим відповісти на питання:

- Яка складність алгоритму за часом та пам’яттю?

- Чому час роботи не залежить від вхідних даних?

- У чому ключова перевага сортування вибором над бульбашковим?

- Чи є алгоритм стійким? Чому?

Тести

1. Яка часова складність сортування вибором для вже відсортованого масиву?

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) O(1)

2. Яку максимальну кількість обмінів виконає сортування вибором для масиву з 10 елементів?

1) 9

2) 10

3) 45

4) 100

3. Основний недолік сортування вибором полягає у:

1) Великій кількості операцій обміну.

2) Високих вимогах до пам’яті (не є in-place).

3) Нестійкості.

4) Великій кількості порівнянь, незалежно від упорядкованості масиву.

4. Для якої з наступних ситуацій сортування вибором є найбільш доцільним?

1) Сортування масиву з мільйона цілих чисел.

2) Сортування списку імен користувачів для виведення на веб-сторінку.

3) Сортування масиву, де операції запису в пам’ять є дуже повільними.

4) Сортування майже відсортованого масиву.

5. Розглянемо масив `[3, 5, 1, 2, 4]`. Яким буде стан масиву після двох повних проходів (ітерацій `i=0` та `i=1`) сортування вибором?

1) `[1, 2, 3, 5, 4]`

2) `[1, 5, 3, 2, 4]`

3) `[1, 2, 5, 3, 4]`

4) `[3, 1, 5, 2, 4]`

Відповіді:

1. c)
2. a)
3. d)
4. c)
5. c)

Тести підвищеної складності

1. Якщо реалізувати сортування вибором, яке на кожній ітерації шукає максимальний елемент і ставить його в кінець невідсортованої частини, як це вплине на загальну продуктивність?

1) Алгоритм стане значно швидшим.

2) Алгоритм стане значно повільнішим.

3) Асимптотична складність та кількість операцій не зміняться.

4) Алгоритм стане стійким.

2. Скільки порівнянь виконає сортування вибором для масиву з 8 елементів?

1) 8

2) 28

3) 36

4) 64

3. Яким буде кінцевий результат сортування масиву `[6, 3, 5a, 2, 5b]` (де `5a` і `5b` рівні за значенням) за допомогою сортування вибором?

1) `[2, 3, 5a, 5b, 6]`

2) `[2, 3, 5b, 5a, 6]`

3) `[2, 3, 6, 5a, 5b]`

4) `[2, 6, 5a, 3, 5b]`

4. Якщо час однієї операції порівняння дорівнює 1 нс, а час операції обміну – 100 нс, для якого з алгоритмів сортування масиву `[5, 4, 3, 2, 1]` загальний час буде меншим?

1) Бульбашкове сортування.

2) Сортування вибором.

3) Час буде приблизно однаковим.

4) Неможливо визначити.

5. Сортування вибором виконує $O(n^2)$ порівнянь та $O(n)$ обмінів. Сортування вставками (Insertion Sort) у найгіршому випадку виконує $O(n^2)$ порівнянь та $O(n^2)$ обмінів. В якому випадку сортування вибором буде однозначно кращим?

1) Сортування масиву, що вже відсортований.

2) Сортування масиву з унікальними елементами.

3) Сортування масиву великих об’єктів, де обмін є дорогою операцією.

4) Сортування масиву, відсортованого у зворотному порядку.

6. Яка мінімальна кількість обмінів, яку може виконати сортування вибором на масиві з `n` елементів?

1) 0

2) 1

3) `n-1`

4) `n/2`

7. Розглянемо “подвійне сортування вибором”, яке за один прохід знаходить і мінімальний, і максимальний елементи, і ставить їх на початок і в кінець відповідно. Як це вплине на кількість порівнянь?

1) Кількість порівнянь зменшиться приблизно вдвічі.

2) Кількість порівнянь не зміниться.

3) Кількість порівнянь зросте приблизно вдвічі.

4) Кількість порівнянь зменшиться до $O(n \log n)$.

8. Якщо застосувати сортування вибором до зв'язного списку (singly linked list), яка операція стане "вузьким місцем" і погіршить продуктивність порівняно з масивом?

1) Пошук мінімального елемента.

2) Операція обміну елементів.

3) Переміщення до `i`-го елемента.

4) Жодна, складність залишиться такою ж.

9. Який інваріант циклу є правильним для зовнішнього циклу сортування вибором (на початок ітерації `i`)?

1) Підмасив `A[0...i-1]` містить `i` найменших елементів всього масиву у відсортованому порядку.

2) Підмасив `A[0...i-1]` містить елементи, менші за будь-який елемент в `A[i...n-1]`.

3) Елемент `A[i]` є найменшим у підмасиві `A[i...n-1]`.

4) Усі елементи в `A[0...i-1]` менші за `A[i]`.

10. Скільки обмінів виконає сортування вибором на масиві `[3, 5, 2, 8, 5, 4]`?

1) 3

2) 4

3) 5

4) 6

Відповіді:

1. 3) Асимптотична складність та кількість операцій не зміняться. Логіка залишається дзеркальною, кількість порівнянь та обмінів буде такою ж.

2. 2) 28. Кількість порівнянь = $n(n-1)/2 = 8 \times 7/2 = 28$.

3. 1) 1) min=2, swap(6,2) -> `[2,3,5a,6,5b]`. 2) min=3 -> no swap. 3) min=5a, scan... 5b=5a, `min_idx` не зміниться, swap(5a,5a). 4) min=5b, swap(6,5b) -> `[2,3,5a,5b,6]`. Відповідь а) є правильною, якщо `min_idx` не оновлюється

для рівних елементів. Якщо оновлюється, то b). Класична реалізація `A[j] < A[min_idx]` робить його стійким для таких випадків, отже 1) - вірна.

4. 2) Сортування вибором. К-сть порівнянь $5 \times 4/2 = 10$. К-сть обмінів = 4. Час: $10 \times 1 + 4 \times 100 = 410$ нс. Для Bubble Sort к-сть обмінів у гіршому випадку = 10. Час: $10 \times 1 + 10 \times 100 = 1010$ нс.

5. 3) Сортування масиву великих об'єктів, де обмін є дорогою операцією. Це основна перевага сортування вибором - мінімальна кількість обмінів ($O(n)$).

6. 1) 0. Якщо масив вже відсортований, `min_idx` завжди буде дорівнювати `i`, і умова `min_idx != i` ніколи не виконається.

7. 1) Кількість порівнянь зменшиться приблизно вдвічі. Замість $n-1$ проходів, буде $n/2$ проходів, але кожен прохід буде складнішим. Загальна кількість порівнянь буде близько $1.5n$ за прохід, але асимптотика залишиться $O(n^2)$. Проте кількість проходів циклу зменшиться.

8. 2) Операція обміну елементів. У масиві обмін - це $O(1)$. У зв'язному списку, щоб поміняти два вузли, не знаючи їхніх попередників, потрібно пройти список знову, що робить обмін операцією $O(n)$.

9. 1) Підмасив `A[0...i-1]` містить `i` найменших елементів всього масиву у відсортованому порядку. Це є основним інваріантом, який підтримується на кожному кроці.

10. 3) 5. 1) min=2, swap(3,2) `[2,5,3,8,5,4]`. 2) min=3, swap(5,3) `[2,3,5,8,5,4]`. 3) min=4, swap(5,4) `[2,3,4,8,5,5]`. 4) min=5, swap(8,5) `[2,3,4,5,8,5]`. 5) min=5, swap(8,5) `[2,3,4,5,5,8]`. Всього 5 обмінів.

Завдання для самостійної роботи

1. Реалізуйте сортування вибором, яке сортує масив за спаданням.

2. Напишіть функцію, яка реалізує стійку версію сортування вибором. (Підказка: замість обміну, зсувайте елементи). Проаналізуйте складність такої реалізації.

3. Реалізуйте "подвійне сортування вибором" (знаходження мінімуму та максимуму за один прохід). Порівняйте його продуктивність (кількість порівнянь, обмінів та загальний час) з класичною версією.

4. Порівняйте на практиці час сортування для масиву великих структур та масиву простих цілих чисел. Чи підтверджується теорія про перевагу сортування вибором при дорогих обмінах?

5. Напишіть програму, що візуалізує роботу сортування вибором, виділяючи кольором відсортовану частину, поточний елемент для обміну та знайдений мінімум.

6. Адаптуйте алгоритм для сортування масиву рядків за алфавітом.

7. Дослідіть, як наявність великої кількості дублікатів у масиві впливає на роботу алгоритму.

8. Реалізуйте сортування вибором для двовимірного масиву (матриці) за елементами головної діагоналі.

9. Напишіть тести, які перевіряють нестійкість вашої реалізації сортування вибором.

10. Проаналізуйте використання кеш-пам'яті сортуванням вибором. Чи є його патерн доступу до пам'яті дружнім до кешу? Порівняйте з бульбашковим сортуванням.

Теми лабораторних робіт

1. Порівняльний аналіз кількості операцій запису в пам'ять для алгоритмів `Selection Sort`, `Bubble Sort` та `Insertion Sort`.

2. Розробка та тестування стійкої версії сортування вибором та аналіз її накладних витрат.

3. Реалізація та аналіз "Подвійного сортування вибором" (`MinMax Sort`).

4. Застосування сортування вибором для сортування зовнішніх даних (даних, що не вміщуються в оперативну пам'ять), де читання є дешевим, а запис - дорогим.

5. Аналіз продуктивності сортування вибором на різних структурах даних: масив, двобічно зв'язний список.

6. Створення гібридного сортування: використання `MergeSort` до певного рівня рекурсії з переходом на `SelectionSort` для малих підмасивів.

7. Розробка програми-бенчмарку для вимірювання продуктивності простих сортувань на різних типах даних (випадкові, відсортовані, з дублікатами).

8. Візуалізатор патернів доступу до пам'яті для простих алгоритмів сортування.

9. Дослідження асимптотичної поведінки: графічне підтвердження, що час роботи сортування вибором зростає пропорційно n^2.

10. Застосування сортування вибором в криптографічному контексті, де важливо мати передбачуваний час виконання, щоб уникнути атак по часу.

9.1.3 Сортування вставкою

Теоретичні відомості

Сортування вставкою — це простий та ефективний для невеликих наборів даних алгоритм сортування. Його робота імітує спосіб, у який багато людей сортують карти в руках: ви тримаєте вже відсортовану частину карт, берете нову карту і вставляєте її у правильне місце серед відсортованих.

Алгоритм ділить масив на дві частини: відсортовану, яка знаходиться на початку масиву, і невідсортовану. На кожному кроці алгоритм бере перший елемент з невідсортованої частини і вставляє його на правильну позицію у відсортованій частині, зсуваючи інші елементи праворуч, щоб звільнити місце.

Основний принцип роботи:

1. Алгоритм починає з другого елемента (індекс 1), вважаючи, що перший елемент (індекс 0) сам по собі вже є відсортованим підмасивом.

2. На кожній ітерації `i` (від 1 до `n-1`), елемент `A[i]` (назвемо його `key`) виймається.

3. Далі алгоритм шукає правильну позицію для `key` у відсортованому підмасиві `A[0...i-1]`. Пошук ведеться справа наліво.

4. Поки елементи у відсортованій частині більші за `key`, вони зсуваються на одну позицію праворуч.

5. Коли знаходиться елемент, що є меншим або рівним `key`, або досягнуто початок масиву, `key` вставляється на звільнене місце.

6. Цей процес повторюється, доки всі елементи не будуть переміщені з невідсортованої частини у відсортовану.

Псевдокод алгоритму

```
function InsertionSort(array A)
  n = length(A)
  // Зовнішній цикл, що ітерує по невідсортованій частині.
  // Починаємо з другого елемента (i=1), оскільки перший елемент вваж
ається відсортованим.
  for i from 1 to n-1
    // key - це поточний елемент, який ми хочемо вставити у відсортов
```

```
ану частину.
    key = A[i]

    // j - індекс останнього елемента у відсортованій частині.
    j = i - 1

    // Внутрішній цикл: зсуваємо елементи відсортованої частини, які
більші за key,
    // на одну позицію праворуч, щоб звільнити місце для вставки.
    while j >= 0 and A[j] > key
      A[j+1] = A[j]
      j = j - 1
    end while

    // Вставляємо key на правильну позицію.
    A[j+1] = key
  end for
end function
```

- **A**: Масив для сортування.

- **n**: Кількість елементів.

- **i**: Індекс першого елемента в невідсортованій частині.

- **key**: Значення елемента **A[i]**, яке потрібно вставити.

- **j**: Індекс для сканування відсортованої частини справа наліво.

Приклад роботи

Розглянемо сортування масиву **A = [5, 2, 4, 6, 1, 3]**.

- **i = 1: key = 2**. Відсортована частина **[5]**.

- **j=0**. **A[0] (5) > key (2)**. Зсуваємо **5** праворуч: **A[1] = 5**. Масив: **[5, 5, 4, 6, 1, 3]**.

- **j** стає **-1**. Цикл завершено.

- Вставляємо **key** на позицію **j+1** (0).

- Масив: **[**2, 5**, 4, 6, 1, 3]**.

- **i = 2: key = 4**. Відсортована частина **[2, 5]**.

- **j=1**. **A[1] (5) > key (4)**. Зсуваємо **5** праворуч. Масив: **[2, 5, 5, 6, 1, 3]**.

- **j=0**. **A[0] (2) < key (4)**. Цикл завершено.

- Вставляємо **key** на позицію **j+1** (1).

- Масив: `[**2, 4, 5**, 6, 1, 3]`.

- **i** = 3: `key = 6`. Відсортована частина `[2, 4, 5]`.

- `j=2`. `A[2] (5) < key (6)`. Цикл не виконується.

- Вставляємо `key` на позицію `j+1` (3).

- Масив: `[**2, 4, 5, 6**, 1, 3]`.

- **i** = 4: `key = 1`. Відсортована частина `[2, 4, 5, 6]`.

- `j=3`. `A[3](6)>1`, зсув. `j=2`. `A[2](5)>1`, зсув. `j=1`. `A[1](4)>1`, зсув. `j=0`. `A[0](2)>1`, зсув.

- `j` стає `-1`. Цикл завершено.

- Вставляємо `key` на позицію `j+1` (0).

- Масив: `[**1, 2, 4, 5, 6**, 3]`.

- **i** = 5: `key = 3`. Відсортована частина `[1, 2, 4, 5, 6]`.

- `j=4,3,2`. Зсуваємо `6, 5, 4`.

- `j=1`. `A[1](2) < key(3)`. Цикл завершено.

- Вставляємо `key` на позицію `j+1` (2).

- Масив: `[**1, 2, 3, 4, 5, 6**]`.

Аналіз складності

- **Часова складність:**

- Найгірший випадок: O(n^2). Це відбувається, коли масив відсортований у зворотному порядку. Щоб вставити **i**-й елемент, потрібно виконати `i-1` порівнянь і зсувів. Загальна кількість операцій: $1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$, що є $O(n^2)$.

- Середній випадок: O(n^2). Для випадкового масиву в середньому потрібно переглянути половину відсортованої частини, що також призводить до квадратичної складності.

- Найкращий випадок: O(n). Це ключова перевага. Якщо масив вже відсортований, внутрішній цикл `while` ніколи не виконується. Алгоритм просто проходить по масиву один раз, роблячи `n-1` порівнянь.

- Просторова складність: O(1). Це “in-place” алгоритм, який потребує лише константного обсягу додаткової пам’яті для зберігання `key` та індексів.

Властивості алгоритму

- Стійкість: Алгоритм є стійким. Умова `A[j] > key` гарантує, що елементи з однаковими значеннями не будуть мінятися місцями. Зсув відбувається тільки для строго більших елементів.

- Адаптивність: Алгоритм є адаптивним, оскільки його продуктивність покращується, якщо масив вже частково відсортований. Найкращий випадок $O(n)$ є яскравим прикладом цього.

- Онлайновість (Online): Сортування вставкою може сортувати дані по мірі їх надходження, не вимагаючи наявності всього набору даних одразу.

Оптимізація: Сортування вставкою з бінарним пошуком

Пошук місця для вставки `key` у відсортованій частині є лінійним ($O(i)$). Його можна замінити на бінарний пошук ($O(\log i)$). Це зменшує кількість порівнянь, але не змінює загальну складність у найгіршому випадку, оскільки кількість зсувів елементів в масиві все одно залишається $O(i)$. Таким чином, загальна складність залишається $O(n^2)$, але ця оптимізація може бути корисною, якщо операція порівняння є значно дорожчою за операцію присвоєння.

Область застосування

Незважаючи на квадратичну складність, сортування вставкою є надзвичайно корисним і широко застосовується:

- Сортування малих масивів: Для невеликих `n` (зазвичай, $n \< 16$), сортування вставкою часто є найшвидшим через свою простоту та низькі накладні витрати.

- Сортування майже відсортованих даних: Завдяки своїй адаптивності, воно є ідеальним вибором, коли дані вже частково впорядковані.

- Онлайн-алгоритми: Коли дані надходять потоком і їх потрібно сортувати “на льоту”.

- Компонент складних алгоритмів: Сортування вставкою є ключовою частиною багатьох гібридних алгоритмів, таких як Timsort (використовується в Python та Java) та Introsort, які переходять на сортування вставкою для обробки малих підмасивів.

Приклади застосування

Уявіть, що ви розробляєте систему, яка показує список завдань, відсортований за пріоритетом. Коли користувач додає нове завдання, його потрібно вставити у вже відсортований список. Замість того, щоб

пересортовувати весь список заново, можна використати логіку сортування вставкою, щоб швидко знайти правильне місце для нового завдання та вставити його.

Переваги та недоліки

Переваги:

- Простота реалізації.

- Висока ефективність для малих наборів даних.

- Адаптивність: найкраща часова складність $O(n)$ на вже відсортованих або майже відсортованих даних.

- Стійкість: зберігає відносний порядок однакових елементів.

- Онлайновість: може обробляти дані по мірі їх надходження.

- Ефективність по пам’яті: просторова складність `O(1)`.

Недоліки:

- Низька ефективність на великих масивах: часова складність $O(n^2)$ робить його непридатним для сортування великих обсягів невпорядкованих даних.

- Велика кількість зсувів: у найгіршому випадку виконує $O(n^2)$ операцій зсуву елементів.

Контрольні запитання

1. Поясніть, чому сортування вставкою має часову складність $O(n)$ у найкращому випадку, на відміну від сортування вибором, яке завжди має $O(n^2)$.

2. Наведіть приклад масиву, для якого сортування вставкою виконає максимальну кількість зсувів.

3. Чому сортування вставкою є стійким? Поясніть на прикладі з однаковими елементами.

4. В чому полягає ідея оптимізації сортування вставкою за допомогою бінарного пошуку? Чому ця оптимізація не змінює асимптотичну складність у найгіршому випадку для масивів?

5. Порівняйте сортування вставкою та сортування вибором. У яких ситуаціях кожен з них буде кращим вибором?

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз адаптивності алгоритму сортування вставкою.

2. Мета:

- Реалізувати алгоритм сортування вставкою.

- Експериментально дослідити його продуктивність на різних типах даних.

- Наочно продемонструвати адаптивність алгоритму, порівнявши час роботи на відсортованих, зворотних та випадкових масивах.

3. Завдання:

- Реалізувати функцію сортування вставкою.

- Створити функції для генерації масивів: випадкового, відсортованого, відсортованого у зворотному порядку та майже відсортованого (наприклад, відсортований масив з 5% переставлених елементів).

- Написати програму для вимірювання часу виконання сортування на кожному з типів даних для масивів різного розміру.

- Побудувати графіки залежності часу виконання від розміру масиву для кожного типу даних.

4. Опис: Лабораторна робота фокусується на ключовій перевазі сортування вставкою — його адаптивності. Студент повинен не просто реалізувати алгоритм, а й провести серію експериментів, які чітко покажуть, наскільки швидше алгоритм працює на майже впорядкованих даних порівняно з хаотичними або зворотними даними.

5. Порядок виконання:

- Створити проект та реалізувати функцію `InsertionSort`.

- Реалізувати генератори масивів для чотирьох зазначених випадків.

- У головній програмі створити цикли для тестування на масивах різного розміру (наприклад, 100, 1000, 5000, 10000, 20000 елементів).

- Для кожного розміру та кожного типу даних запускати сортування, вимірюючи час виконання. Рекомендується проводити кілька замірів для усереднення результату.

- Зібрати дані у таблицю.

- На основі даних побудувати 4 криві на одному графіку, що показують залежність часу від розміру n.

- Проаналізувати графік та зробити висновки.

6. Результат:

- Програма, що реалізує сортування вставкою та проводить заміри продуктивності.

- Звіт з графіком, що наочно демонструє часову складність $O(n)$ для відсортованих даних та $O(n^2)$ для зворотних та випадкових.

- Висновки, що пояснюють отримані результати та описують практичне значення адаптивності.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:

    - Функція сортування вставкою.

    - Функції генерації тестових наборів даних.

- Продемонструвати роботу додатку:

    - Запуск сортування для невеликого масиву з виведенням проміжних станів.

    - Запуск вимірювань та демонстрація різниці у часі для відсортованого та зворотного масивів однакового розміру.

- Бути готовим відповісти на питання:

    - Яка складність алгоритму в найкращому, середньому та найгіршому випадках?

    - Що робить цей алгоритм адаптивним?

    - Чому сортування вставкою використовується як частина Timsort?

    - Чи є алгоритм стійким? Поясніть чому.

Тести

1. Який випадок є найкращим для сортування вставкою?

1) Масив, відсортований у зворотному порядку.

2) Масив, заповнений випадковими числами.

3) Масив, який вже повністю відсортований.

4) Масив, що складається з однакових елементів.

2. Яка ключова властивість робить сортування вставкою корисним для гібридних алгоритмів, таких як Timsort?

1) Його Стійкість.

2) Просторова складність O(1).

3) Його висока ефективність на невеликих або майже відсортованих масивах.

4) Простота реалізації.

3. Розглянемо масив `[7, 3, 5, 2]`. Яким буде стан масиву після завершення ітерації зовнішнього циклу для `i = 2` (коли `key=5`)?

1) `[3, 7, 5, 2]`

2) `[3, 5, 7, 2]`

3) `[7, 3, 5, 2]`

4) `[2, 3, 5, 7]`

4. Чому сортування вставкою є “онлайн” алгоритмом?

1) Тому що його код можна легко знайти в Інтернеті.

2) Тому що він може сортувати дані по мірі їх надходження, не чекаючи на повний набір.

3) Тому що він використовує лише одну додаткову змінну.

4) Тому що він працює швидше за інші прості сортування.

5. Сортування вставкою з бінарним пошуком зменшує кількість порівнянь. Чому загальна складність у найгіршому випадку залишається $O(n^2)$?

1) Бінарний пошук сам по собі має складність $O(n^2)$.

2) Кількість операцій зсуву елементів у масиві залишається $O(n)$.

3) Це помилка, складність покращується до $O(n \log n)$.

4) Через високі вимоги до пам’яті.

Відповіді:

1. c) (також d, але c є більш загальним і правильним випадком)
2. c)
3. b)
4. b)
5. b)

Тести підвищеної складності

1. Скільки операцій зсуву (не порівнянь) виконає сортування вставкою на масиві `[5, 4, 3, 2, 1]`?

1) 5

2) 15

3) 10

4) 4

2. Який алгоритм ви б обрали для сортування списку з 10 мільйонів записів, який оновлюється щосекунди додаванням 5 нових записів?

1) Сортування вибором, щоб мінімізувати кількість записів.

2) QuickSort, щоб швидко пересортовувати весь список щоразу.

3) Сортування вставкою, щоб ефективно додавати нові елементи до вже відсортованого списку.

4) Бульбашкове сортування, бо воно найпростіше.

3. Якщо реалізувати сортування вставкою для однозв'язного списку, як зміниться його асимптотична складність у найгіршому випадку?

1) Залишиться $O(n^2)$.

2) Покращиться до $O(n \log n)$, бо вставка у список — це $O(1)$.

3) Погіршиться до $O(n^3)$.

4) Залишиться $O(n^2)$, але може бути ефективнішим на практиці, ніж версія для масиву.

4. Розглянемо масив **A**. Якщо відомо, що кожен елемент знаходиться не далі ніж на **k** позицій від свого фінального відсортованого місця, яка буде часова складність сортування вставкою?

1) $O(n^2)$

2) $O(nk)$

3) $O(n \log k)$

4) $O(n + k)$

5. Яка головна перевага сортування вставкою над бульбашковим сортуванням на майже відсортованих даних?

1) Сортування вставкою стабільне, а бульбашкове — ні.

2) Сортування вставкою виконує менше порівнянь, оскільки може зупинитися раніше.

3) Сортування вставкою використовує менше пам'яті.

4) Сортування вставкою виконує менше обмінів.

6. Скільки порівнянь у внутрішньому циклі (`while`) буде виконано для сортування масиву `[1, 2, 3, 5, 4]`?

1) 1

2) 2

3) 3

4) 5

7. Чим сортування вставкою принципово відрізняється від сортування вибором?

1) Сортування вставкою порівнює сусідні елементи, а вибором — ні.

2) Сортування вставкою будує відсортований підмасив, вставляючи елементи по одному; сортування вибором знаходить наступний мінімальний елемент і ставить його на місце.

3) Сортування вставкою є `in-place`, а вибором — ні.

4) Сортування вставкою нестабільне, а вибором — стабільне.

8. Рекурсивна реалізація сортування вставкою для масиву `A` розміром `n` зазвичай виглядає так: `InsertionSort(A, n-1)`, а потім вставити `A[n-1]` у відсортований підмасив `A[0...n-2]`. Якою буде глибина рекурсії?

1) O(log n)

2) O(n)

3) $O(n^2)$

4) O(1)

9. Який інваріант циклу є правильним для зовнішнього циклу сортування вставкою (на початок ітерації `i`)?

1) Підмасив `A[0...i-1]` є відсортованим.

2) Підмасив `A[0...i-1]` містить `i` найменших елементів з усього масиву.

3) Всі елементи в `A[0...i-1]` менші за всі елементи в `A[i...n-1]`.

4) Елемент `A[i]` є найбільшим у масиві.

10. Який з наведених алгоритмів НЕ є адаптивним?

1) Оптимізоване бульбашкове сортування.

2) Сортування вставкою.

3) Сортування вибором.

4) Timsort.

Відповіді:

1. 3) 10. Для `5` 0 зсувів. Для `4` 1 зсув. Для `3` 2 зсуви. Для `2` 3 зсуви. Для `1` 4 зсуви. Всього: $0 + 1 + 2 + 3 + 4 = 10$.

2. 3) Сортування вставкою, щоб ефективно додавати нові елементи до вже відсортованого списку. Це ідеальний приклад онлайн-сортування.

3. 4) Залишиться $O(n^2)$, але може бути ефективнішим на практиці, ніж версія для масиву. Пошук місця для вставки все ще $O(n)$, але сама вставка (зміна вказівників) — це $O(1)$, на відміну від $O(n)$ зсувів у масиві.

4. 2) O(nk). Для кожного з `n` елементів внутрішній цикл виконається не більше `k` разів.

5. 2) Сортування вставкою виконує менше порівнянь, оскільки може зупинитися раніше. Оптимізоване бульбашкове сортування пройде весь масив до кінця, тоді як вставка зупиниться, як тільки знайде своє місце.

6. 1) 1. Порівняння будуть тільки на останньому елементі `key=4`. `(5 > 4)` - так, зсув. `(3 < 4)` - ні, зупинка. Одне порівняння у циклі while, яке призвело до зсуву.

7. 2) Сортування вставкою будує відсортований підмасив, вставляючи елементи по одному; сортування вибором знаходить наступний мінімальний елемент і ставить його на місце. Це фундаментальна різниця в їхній стратегії.

8. 2) O(n). Рекурсія буде викликана `n-1` разів, створюючи стек викликів глибиною `n`.

9. 1) Підмасив `A[0...i-1]` є відсортованим. Це ключова умова, яка підтримується на кожному кроці.

10. 3) Сортування вибором. Його час роботи не залежить від початкової впорядкованості даних, він завжди виконує $O(n^2)$ порівнянь.

Завдання для самостійної роботи

1. Реалізуйте сортування вставкою для двобічно зв’язного списку (doubly linked list). Порівняйте його продуктивність з версією для масиву.

2. Напишіть рекурсивну версію сортування вставкою.

3. Реалізуйте сортування вставкою з бінарним пошуком для знаходження позиції вставки. Проведіть тести, щоб визначити, наскільки швидше ця

версія працює порівняно з класичною (виміряйте час для масивів з дорогими для порівняння об’єктами).

4. Напишіть функцію, яка підраховує точну кількість порівнянь та зсувів для сортування вставкою на заданому масиві.

5. Дослідіть, як сортування вставкою поводиться на масивах з невеликою кількістю унікальних елементів (наприклад, масив з мільйона чисел від 1 до 10).

6. Адаптуйте алгоритм для сортування масиву рядків за їх довжиною, а при однаковій довжині — за алфавітом.

7. Створіть візуалізатор, який показує процес сортування вставкою, виділяючи `key` та процес зсуву елементів.

8. Порівняйте на практиці продуктивність сортування вставкою з оптимізованим бульбашковим сортуванням на різних типах даних.

9. Реалізуйте сортування вставкою “справа наліво”, де відсортована частина масиву накопичується в кінці.

10. Дослідіть алгоритм Shellsort, який є вдосконаленням сортування вставкою, та поясніть, чому він ефективніший.

Теми лабораторних робіт

1. Порівняльний аналіз адаптивності простих сортувань: **`Insertion Sort`** vs. **`Optimized Bubble Sort`**.

2. Реалізація та аналіз продуктивності **`Binary Insertion Sort`**.

3. Застосування сортування вставкою для злиття відсортованих списків в онлайн-режимі.

4. Створення бібліотеки сортування для зв’язних списків, що включає **`Insertion Sort`**.

5. Дослідження та реалізація **`Shellsort`** як узагальнення **`Insertion Sort`**.

6. Розробка міні-версії **`Timsort`**: гібридний алгоритм, що використовує **`Merge Sort`** та **`Insertion Sort`**.

7. Аналіз продуктивності **`Insertion Sort`** на даних, що слідують різним розподілам (рівномірний, нормальний, експоненційний).

8. Створення програми для аналізу **`k-сортованих`** масивів та ефективності **`Insertion Sort`** на них.

9. Візуалізатор для порівняння `Insertion Sort`, `Selection Sort` та `Bubble Sort` на однакових наборах даних.

10. Аналіз кеш-промахів (cache misses) для простих алгоритмів сортування, включаючи `Insertion Sort`.

9.1.4 Сортування Шелла

Теоретичні відомості

Сортування Шелла, назване на честь його винахідника Дональда Шелла, є значним удосконаленням алгоритму сортування вставкою. Основна проблема сортування вставкою полягає в тому, що воно переміщує елементи лише на одну позицію за раз. Це означає, що якщо малий елемент знаходиться в кінці великого масиву, йому доведеться пройти через тисячі зсувів, щоб стати на своє місце, що призводить до квадратичної складності $O(n^2)$.

Сортування Шелла вирішує цю проблему, дозволяючи обмін елементів, що знаходяться далеко один від одного. Головна ідея — порівнювати елементи не лише з сусідами, а з елементами на певній відстані (яку називають проміжком або кроком, англ. gap). Алгоритм поступово зменшує цей проміжок, доки він не стане рівним 1.

Основний принцип роботи: h-сортування

Алгоритм працює в кілька фаз. Кожна фаза характеризується певним значенням проміжку `h`. На кожній фазі масив стає **h-відсортованим**.

Що таке h-сортування? Уявіть, що ми розбиваємо масив на `h` незалежних підмасивів.

- Перший підмасив складається з елементів `A[0], A[h], A[2h], ...`

- Другий підмасив — з `A[1], A[1+h], A[1+2h], ...`

- і так далі до `h`-го підмасиву.

На фазі з проміжком `h` алгоритм сортує кожен з цих `h` підмасивів за допомогою сортування вставкою. Цей процес називається h-сортуванням.

Ключова ідея полягає в тому, що після h-сортування з великим проміжком `h` малі елементи швидко переміщуються на початок масиву, а великі — в кінець. Масив стає “частково впорядкованим”. Коли ми зменшуємо проміжок, масив стає ще більш впорядкованим.

Остання фаза завжди виконується з проміжком `h=1`. На цьому етапі h-сортування — це звичайне сортування вставкою. Однак, оскільки попередні фази вже значно впорядкували масив, цей фінальний прохід є дуже ефективним (близьким до лінійного часу $O(n)$).

Послідовність проміжків

Продуктивність сортування Шелла критично залежить від вибору послідовності проміжків. Це одна з найскладніших та найцікавіших частин алгоритму.

- Оригінальна послідовність Шелла: `n/2, n/4, ..., 1`. Ця послідовність проста, але може призвести до найгіршої часової складності $O(n^2)$, особливо якщо `n` є степенем двійки.

- Послідовність Хіббарда: $2^k - 1$ (`..., 15, 7, 3, 1`). Забезпечує складність $O(n^{3/2})$.

- Послідовність Кнута (1973): $h_i = (3^i - 1)/2$ (`..., 40, 13, 4, 1`). Часто використовується на практиці. Найбільш популярна формула, що генерує схожі проміжки: $h = 3h + 1$. Послідовність генерується доти, доки $h \< n/3$, а потім використовується у зворотному порядку. Це забезпечує середню складність близько $O(n^{1.25})$.

- Послідовність Седжвіка: Існує кілька, наприклад, $4^k + 3 \cdot 2^{k-1} + 1$. Ці послідовності дають одні з найкращих результатів, наближаючи складність до $O(n^{4/3})$ або навіть $O(n^{7/6})$ у деяких випадках.

Для нашого прикладу ми використаємо послідовність Кнута ($h = 3h + 1$), оскільки вона проста в реалізації та дуже ефективна.

Псевдокод алгоритму (з послідовністю Кнута)

```
function ShellSort(array A)
  n = length(A)

  // 1. Генеруємо послідовність проміжків (h = 3h + 1)
  h = 1
  while h < n/3
    h = 3*h + 1
  end while

  // 2. Сортуємо масив, зменшуючи проміжок h
  while h >= 1
    // Це h-сортування. Воно схоже на сортування вставкою, але з пром
іжком h.
```

```
    // Починаємо з h-го елемента
    for i from h to n-1
      // Беремо поточний елемент (key)
      key = A[i]
      j = i

      // Зсуваємо попередні h-відсортовані елементи, поки не знайдемо
      // правильну позицію для key
      while j >= h and A[j-h] > key
        A[j] = A[j-h]
        j = j - h
      end while
      A[j] = key
    end for

    // Зменшуємо проміжок для наступної фази
    h = h / 3
  end while
end function
```

- `h`: Проміжок між елементами, що порівнюються.

- Перший цикл `while` генерує початковий, найбільший проміжок.

- Другий цикл `while` ітерує по послідовності проміжків у зворотному порядку.

- Цикл `for` та вкладений у нього `while` реалізують саме h-сортування, яке є модифікованим сортуванням вставкою.

Приклад роботи

Розглянемо масив `A = [35, 12, 88, 5, 29, 64, 41, 17]`. Розмір `n=8`. Послідовність проміжків Кнута: `h = 1`. `h = 3*1+1=4`. `4 < 8/3` - ні. Отже, почнемо з `h=4`. Потім `h=1`.

- **Фаза 1: h = 4**

- Ми маємо 4 підмасиви для сортування:

1. `A[0], A[4]` -> `(35, 29)`. Сортуємо: `[29, 35]`.

2. `A[1], A[5]` -> `(12, 64)`. Вже відсортовано.

3. `A[2], A[6]` -> `(88, 41)`. Сортуємо: `[41, 88]`.

4. `A[3], A[7]` -> `(5, 17)`. Вже відсортовано.

- Після застосування цих сортувань масив `A` стає: `[29, 12, 41, 5, 35, 64, 88, 17]`.

- Зверніть увагу, малі елементи **29** і **5** значно просунулися вліво.

- **Фаза 2: h = 1**

- Тепер ми застосовуємо звичайне сортування вставкою до масиву **[29, 12, 41, 5, 35, 64, 88, 17]**.

- Оскільки масив вже "майже відсортований", цей крок буде дуже швидким.

- **key=12**: **[12, 29, 41, 5, 35, 64, 88, 17]**

- **key=41**: на місці.

- **key=5**: **[5, 12, 29, 41, 35, 64, 88, 17]**

- ... і так далі.

- Кінцевий результат: **[5, 12, 17, 29, 35, 41, 64, 88]**.

Аналіз складності

- Часова складність: Як зазначалося, вона сильно залежить від послідовності проміжків і є предметом досліджень.

- Найгірший випадок: $O(n^2)$ для поганих послідовностей (напр., **n/2, n/4, ...**). Для добрих послідовностей (Кнута, Седжвіка) — $O(n^{4/3})$ або $O(n^{3/2})$.

- Середній випадок: Для хороших послідовностей аналітично не доведений, але емпірично близький до $O(n^{1.25})$ або $O(n^{7/6})$.

- Найкращий випадок: $O(n\log n)$ для деяких послідовностей на певних даних, або $O(n)$ якщо масив вже h-відсортований для всіх **h**.

- Просторова складність: O(1). Це одна з ключових переваг. Алгоритм є "in-place".

Властивості алгоритму

- Стійкість: Алгоритм є нестійким. Обмін елементів на великих відстанях може легко порушити відносний порядок однакових елементів.

- Адаптивність: Лише частково. Його продуктивність не так сильно залежить від початкової впорядкованості, як у чистого сортування вставкою, але фінальний прохід з **h=1** виграє від попередніх фаз.

Область застосування

Сортування Шелла — це чудовий вибір "робочої конячки" для сортування загального призначення.

- Сортування масивів середнього розміру: Ідеально підходить для масивів від кількох тисяч до десятків тисяч елементів, де ефективність $O(n^2)$ вже

неприйнятна, а накладні витрати рекурсивних $O(n\log n)$ алгоритмів (як-от QuickSort) можуть бути значними.

- Вбудовані системи та середовища з обмеженою пам'яттю: Завдяки просторовій складності $O(1)$ сортування Шелла є чудовим вибором там, де не можна виділити додаткову пам'ять для сортування (на відміну від MergeSort).

- Заміна простим сортуванням: Це простий спосіб отримати значний приріст продуктивності порівняно з Bubble Sort, Selection Sort або Insertion Sort з мінімальним ускладненням коду.

Приклади застосування

Уявімо прошивку для мережевого маршрутизатора, яка повинна сортувати таблицю активних з'єднань (кілька тисяч записів) за певним критерієм. Використання MergeSort потребувало б додаткової пам'яті, якої може не бути. Використання QuickSort несе ризик $O(n^2)$ у найгіршому випадку, що може "підвісити" пристрій. Сортування Шелла тут є безпечним, швидким та ефективним за пам'яттю вибором.

Переваги та недоліки

Переваги:

- Значно швидше за прості $O(n^2)$ алгоритми.

- Ефективність по пам'яті: O(1).

- Відносно проста реалізація порівняно з QuickSort, MergeSort або HeapSort.

- Відсутність найгіршого випадку $O(n^2)$ при використанні хороших послідовностей проміжків.

Недоліки:

- Нестійкість: не підходить для задач, де потрібно зберігати відносний порядок однакових елементів.

- Складний математичний аналіз: точну часову складність важко передбачити.

- Повільніше за найкращі $O(n\log n)$ алгоритми (QuickSort, MergeSort) на великих масивах даних.

Контрольні запитання

1. У чому полягає головна ідея сортування Шелла, яка дозволяє йому бути швидшим за сортування вставкою?

2. Що таке “h-сортування” і як воно працює?

3. Чому вибір послідовності проміжків є критично важливим для продуктивності алгоритму? Наведіть приклад хорошої та поганої послідовності.

4. Чому сортування Шелла є нестійким? Наведіть невеликий приклад.

5. Чому останній крок сортування Шелла з проміжком `h=1` є ефективним?

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз продуктивності сортування Шелла з різними послідовностями проміжків.

2. Мета:

- Реалізувати алгоритм сортування Шелла.

- Дослідити, як вибір послідовності проміжків впливає на продуктивність алгоритму.

- Порівняти продуктивність сортування Шелла з сортуванням вставкою на різних типах даних.

3. Завдання:

- Реалізувати функцію сортування Шелла.

- Реалізувати алгоритм з як мінімум двома різними послідовностями проміжків (наприклад, оригінальна послідовність Шелла `n/2` та послідовність Кнута `3h+1`).

- Створити функції для генерації масивів: випадкового та відсортованого у зворотному порядку.

- Написати програму, яка вимірює час виконання для обох реалізацій на масивах різного розміру.

- Порівняти отримані результати з часом роботи сортування вставкою на тих же даних.

4. Опис: Ця лабораторна робота фокусується на найважливішому аспекті сортування Шелла — послідовності проміжків. Студент повинен на практиці переконатися, наскільки сильно вибір цієї послідовності впливає на швидкість сортування, та порівняти отриманий результат з базовим алгоритмом (сортуванням вставкою), який Shellsort вдосконалює.

5. Порядок виконання:

- Створити проект та реалізувати функцію `ShellSort`, яка приймає масив та тип послідовності проміжків як параметри.

- Реалізувати генерацію проміжків для послідовності Шелла та Кнута.

- У головній програмі тестувати алгоритм на масивах значного розміру (наприклад, 10 000, 50 000, 100 000, 500 000 елементів).

- Для кожного розміру та типу послідовності виміряти час виконання.

- Провести аналогічні заміри для сортування вставкою, щоб побачити різницю.

- Результати звести в таблицю та/або побудувати графіки залежності часу від розміру `n`.

- Проаналізувати результати та зробити висновки.

6. Результат:

- Програма, що реалізує сортування Шелла з різними стратегіями вибору проміжків.

- Звіт, що містить порівняльні дані продуктивності, графіки та висновки щодо ефективності різних послідовностей проміжків.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:

    - Функція сортування Шелла.

    - Логіка генерації різних послідовностей проміжків.

- Продемонструвати роботу додатку:

    - Запуск сортування з різними послідовностями на одному й тому ж масиві для демонстрації різниці в часі.

    - Порівняння швидкості з сортуванням вставкою на великому масиві.

- Бути готовим відповісти на питання:

    - Яка часова та просторова складність алгоритму?

    - Чому оригінальна послідовність Шелла (`n/2`) може бути неефективною?

    - Чому алгоритм не є стійким?

    - У яких випадках варто віддати перевагу сортуванню Шелла перед QuickSort?

Тести

1. Яке твердження найкраще описує ідею сортування Шелла?

1) Воно рекурсивно ділить масив на дві частини, як QuickSort.

2) Воно знаходить мінімальний елемент і ставить його на правильне місце, як сортування вибором.

3) Воно сортує елементи на великих відстанях один від одного, поступово зменшуючи ці відстані.

4) Воно завжди порівнює тільки сусідні елементи, як бульбашкове сортування.

2. Продуктивність сортування Шелла найбільше залежить від:

1) Кількісті елементів у масиві.

2) Типу даних у масиві (цілі числа, рядки).

3) Обраної послідовності проміжків (gap sequence).

4) Початкової впорядкованості масиву.

3. Яка властивість НЕ притаманна сортуванню Шелла?

1) Просторова складність O(1).

2) Швидше за сортування вставкою на великих масивах.

3) Стійкість.

4) Є узагальненням сортування вставкою.

4. Якою буде часова складність сортування Шелла в найгіршому випадку при використанні послідовності проміжків $h = \lfloor n/2^k \rfloor$?

1) O(n log n)

2) O(n)

3) $O(n^2)$

4) $O(n^{1.5})$

5. Останній прохід сортування Шелла завжди виконується з проміжком **h=1**. Чому цей крок зазвичай дуже швидкий?

1) Тому що масив до цього моменту вже гарантовано відсортований.

2) Тому що попередні проходи з більшими проміжками зробили масив майже відсортованим.

3) Тому що сортування вставкою завжди має складність O(n).

4) Тому що на останньому кроці порівнюється мала кількість елементів.

Відповіді:

1. c)
2. c)
3. c)
4. c)
5. b)

Тести підвищеної складності

1. Масив `[10, 20, 30, 40, 50, 60, 70, 80, 90, 1]` сортується методом Шелла з початковим проміжком `h=3`. Скільки порівнянь буде виконано на цій першій фазі?

1) 0

2) 1

3) 3

4) 10

2. Яка з наведених послідовностей проміжків вважається однією з найефективніших на практиці?

1) `n-1, n-2, ..., 1`

2) `1, 2, 3, 4, ...`

3) `n/2, n/4, ..., 1`

4) `..., 40, 13, 4, 1` (послідовність Кнута)

3. Розглянемо масив `[5a, 2, 3, 5b]`, де `5a` і `5b` рівні за значенням. Яким може бути результат після сортування Шеллом з проміжками `h=2`, а потім `h=1`?

1) `[2, 3, 5a, 5b]`

2) `[2, 3, 5b, 5a]`

3) Обидва варіанти можливі.

4) Алгоритм не відсортує цей масив.

4. Чому сортування Шелла є кращим вибором для вбудованих систем, ніж QuickSort?

1) Shellsort завжди швидший за QuickSort.

2) Shellsort має гарантовану просторову складність O(1), тоді як QuickSort потребує O(log n) пам’яті для стеку рекурсії.

3) Shellsort є стійким алгоритмом.

4) Shellsort легше реалізувати рекурсивно.

5. Якщо масив вже h-відсортований для проміжку `h`, чи залишиться він h-відсортованим після k-сортування з `k < h`?

1) Так, завжди.

2) Ні, не обов’язково.

3) Тільки якщо `h` є кратним `k`.

4) Тільки якщо масив був спочатку відсортований.

6. Що є головним недоліком сортування Шелла порівняно з MergeSort для великих масивів?

1) Shellsort вимагає більше пам’яті.

2) MergeSort є стійким, а Shellsort — ні.

3) Shellsort має гіршу асимптотичну складність у середньому випадку ($O(n^{1.25})$ проти $O(n\log n)$).

4) 2) та c) є вірними.

7. При сортуванні масиву `[6, 5, 4, 3, 2, 1]` методом Шелла з проміжками `h=3, 1`, яким буде масив після фази `h=3`?

1) `[3, 2, 1, 6, 5, 4]`

2) `[1, 2, 3, 4, 5, 6]`

3) `[6, 5, 4, 3, 2, 1]` (нічого не зміниться)

4) `[3, 5, 1, 6, 2, 4]`

8. Сортування Шелла можна розглядати як спробу оптимізувати:

1) Кількість порівнянь у сортуванні вибором.

2) Кількість обмінів у бульбашковому сортуванні.

3) Переміщення елементів на великі відстані у сортуванні вставкою.

4) Використання пам’яті у MergeSort.

9. Якщо час порівняння елементів значно перевищує час їх переміщення, яка оптимізація НЕ дасть суттєвого прискорення для сортування Шелла?

1) Вибір кращої послідовності проміжків.

2) Використання бінарного пошуку всередині h-сортування.

3) Запуск алгоритму на більш потужному процесорі.

4) Перехід на сортування вставкою, якщо масив майже відсортований.

10. Яке твердження про сортування Шелла є хибним?

1) Це сортування методом порівняння.

2) Це сортування “на місці” (in-place).

3) Його часова складність у середньому краща за $O(n^2)$.

4) Його часова складність у найгіршому випадку завжди $O(n\log n)$.

Відповіді:

1. 2) 1. Буде порівнюватися лише підмасив з `A[9]=1`. Він порівняється з `A[6]=70`, `A[3]=40`, `A[0]=10`. Елемент `1` стане на місце `10`, але це одна серія порівнянь в рамках одного сортування вставкою для одного елемента `key=1`.

2. 4) `..., 40, 13, 4, 1` (послідовність Кнута). Емпіричні та теоретичні дослідження показують високу ефективність цієї та подібних послідовностей (наприклад, Седжвіка).

3. 3) Обидва варіанти можливі. Оскільки алгоритм нестабільний, відносний порядок `5a` і `5b` не гарантується.

4. 2) Shellsort має гарантовану просторову складність O(1), тоді як QuickSort потребує O(log n) пам’яті для стеку рекурсії. У системах з жорсткими обмеженнями пам’яті це може бути критично.

5. 2) Ні, не обов’язково. Подальше k-сортування може порушити попередню h-впорядкованість, оскільки воно переміщує елементи за іншою логікою.

6. 4) 2) та c) є вірними. MergeSort є стійким і має кращу асимптотику $O(n\log n)$, що робить його привабливішим для дуже великих наборів даних, якщо є додаткова пам’ять.

7. 1) `[3, 2, 1, 6, 5, 4]`. Підмасиви: `(6,3)->(3,6)`; `(5,2)->(2,5)`; `(4,1)->(1,4)`. Результат: збираємо назад `[3, 2, 1, 6, 5, 4]`.

8. 3) Переміщення елементів на великі відстані у сортуванні вставкою. Це пряме вирішення головного недоліку Insertion Sort.

9. 2) Використання бінарного пошуку всередині h-сортування. Бінарний пошук зменшує кількість порівнянь, але не переміщень. Якщо порівняння і так швидкі, виграш буде мінімальним.

10. 4) Його часова складність у найгіршому випадку завжди $O(n\log n)$. Це хибне твердження; складність залежить від проміжків і може сягати $O(n^2)$.

Завдання для самостійної роботи

1. Реалізуйте сортування Шелла з послідовністю проміжків Седжвіка ($4^k + 3 \cdot 2^{k-1} + 1$). Порівняйте її продуктивність з послідовністю Кнута.

2. Напишіть програму, яка візуалізує процес h-сортування, виділяючи кольором поточний підмасив, що сортується.

3. Дослідіть, як сортування Шелла поводиться на масивах з великою кількістю дублікатів.

4. Адаптуйте алгоритм для сортування масиву об’єктів за певним полем.

5. Напишіть тести, які наочно демонструють нестійкість сортування Шелла.

6. Порівняйте на практиці продуктивність сортування Шелла та QuickSort на масивах різного розміру (малий, середній, великий). Визначте точку, після якої QuickSort стає стабільно швидшим.

7. Дослідіть, чи можна зробити сортування Шелла стійким, і яка ціна такої модифікації.

8. Реалізуйте сортування Шелла для зв’язного списку. Проаналізуйте, наскільки це практично та ефективно.

9. Проаналізуйте поведінку алгоритму на масиві, де всі елементи однакові. Якою буде його складність?

10. Спробуйте розробити власну послідовність проміжків і протестувати її ефективність.

Теми лабораторних робіт

1. Емпіричний аналіз продуктивності різних послідовностей проміжків для сортування Шелла.

2. Порівняльний аналіз **`Shellsort`** та **`QuickSort`** на середніх наборах даних в умовах обмеженої пам’яті.

3. Розробка гібридного алгоритму: використання **`QuickSort`** з переходом на **`Shellsort`** для малих підмасивів.

4. Застосування сортування Шелла для сортування даних у файлі (зовнішнє сортування) шляхом роботи з блоками даних.

5. Дослідження впливу патернів даних (майже відсортовані, зворотні, з періодичними послідовностями) на ефективність **`Shellsort`**.

6. Створення інтерактивного візуалізатора, що дозволяє користувачу обирати послідовність проміжків і спостерігати за процесом сортування.

7. Аналіз кеш-ефективності `Shellsort`: підрахунок кеш-промахів та порівняння з іншими алгоритмами.

8. Реалізація паралельної версії сортування Шелла для багатоядерних систем (наприклад, незалежне сортування h-підмасивів).

9. Оцінка продуктивності `Shellsort` при сортуванні масивів великих структур, де операції порівняння та переміщення є дорогими.

10. Історичний огляд та еволюція послідовностей проміжків для сортування Шелла: від Дональда Шелла до сучасних досліджень.

9.1.5 Шейкерне сортування

Теоретичні відомості

Шейкерне сортування, також відоме як двонаправлене бульбашкове сортування або `Cocktail Sort`, є варіацією та вдосконаленням класичного бульбашкового сортування. Його назва походить від аналогії з приготуванням коктейлю в шейкері, де рідина рухається туди-сюди.

Основна проблема звичайного бульбашкового сортування полягає в повільному переміщенні малих елементів, що знаходяться в кінці масиву, на їхні правильні позиції на початку. Такі елементи отримали назву “черепахи”, оскільки вони зсуваються вліво лише на одну позицію за один повний прохід. Великі ж елементи (“зайці”) швидко “спливають” в кінець масиву. Шейкерне сортування вирішує проблему “черепах”, переміщуючи елементи в обох напрямках.

Основний принцип роботи:

Алгоритм, як і бульбашкове сортування, багаторазово проходить по масиву, порівнюючи та міняючи місцями сусідні елементи. Однак, на відміну від однонаправлених проходів бульбашкового сортування, шейкерне сортування виконує проходи в обидві сторони:

1. **Прохід зліва направо:** Перша частина ітерації повністю аналогічна бульбашковому сортуванню. Алгоритм рухається від початку до кінця масиву, порівнюючи сусідні елементи і переміщуючи найбільший елемент з поточної невідсортованої частини в її кінець.

2. **Прохід справа наліво:** Одразу після першого проходу алгоритм змінює напрямок. Він рухається від кінця невідсортованої частини до її початку, порівнюючи сусідні елементи і переміщуючи найменший елемент на його правильну позицію на початку.

Ці два проходи (вправо і вліво) складають одну повну ітерацію. Після кожної ітерації межі невідсортованої частини масиву звужуються з обох боків: зліва на один елемент (там тепер стоїть гарантований мінімум) і справа на один елемент (там стоїть гарантований максимум).

Оптимізація

Як і бульбашкове сортування, шейкерне можна оптимізувати.

- Якщо протягом повної ітерації (в обидва боки) не відбулося жодного обміну, масив повністю відсортований, і алгоритм може завершити роботу достроково.

- Більш ефективна оптимізація полягає у відстеженні позиції останнього обміну на кожному проході. Наприклад, під час руху зліва направо ми запам’ятовуємо індекс останнього обміну. Всі елементи праворуч від цієї позиції вже відсортовані, тому наступний прохід справа наліво можна починати з цього індексу. Аналогічно для руху в протилежному напрямку.

Псевдокод алгоритму (з оптимізацією меж)

```
function CocktailShakerSort(array A)
  n = length(A)
  left = 0
  right = n - 1
  swapped = true

  while swapped == true
    // 1. Скидаємо прапорець на початку ітерації
    swapped = false

    // 2. Прохід зліва направо (як у Bubble Sort)
    for i from left to right - 1
      if A[i] > A[i+1]
        swap(A[i], A[i+1])
        swapped = true
      end if
    end for

    // Якщо обмінів не було, масив відсортований
    if swapped == false
```

```
      break
    end if

    // 3. Зменшуємо праву межу, оскільки найбільший елемент вже на мі
сці
    right = right - 1

    // 4. Прохід справа наліво
    for i from right - 1 down to left
      if A[i] > A[i+1]
        swap(A[i], A[i+1])
        swapped = true
      end if
    end for

    // 5. Збільшуємо ліву межу, оскільки найменший елемент вже на міс
ці
    left = left + 1
  end while
end function
```

- `A`: Масив для сортування.

- `n`: Розмір масиву.

- `left`, `right`: Межі невідсортованої частини масиву, які поступово звужуються.

- `swapped`: Прапорець, що сигналізує, чи відбувся хоча б один обмін протягом повної ітерації.

Приклад роботи

Розглянемо сортування масиву `A = [5, 1, 4, 2, 8, 0, 3]`. `left=0`, `right=6`.

- **Ітерація 1:**

- **Прохід вправо:**

- `(5,1)`->swap, `[1,5,4,2,8,0,3]`.

- `(5,4)`->swap, `[1,4,5,2,8,0,3]`.

- `(5,2)`->swap, `[1,4,2,5,8,0,3]`.

- `(8,0)`->swap, `[1,4,2,5,0,8,3]`.

- `(8,3)`->swap, `[1,4,2,5,0,3,8]`. Найбільший елемент `8` на своєму місці.

- `right` стає `5`.

- **Прохід вліво:**

- `(3,0)`->swap, `[1,4,2,5,3,0,8]`.

- ...

- `(4,2)`->swap, `[1,2,4,5,3,0,8]`.

- `(2,1)`->swap, `[1,2,4,5,3,0,8]`.

- Після повного проходу вліво: `[0, 1, 4, 2, 5, 3, 8]`. Найменший елемент `0` на своєму місці.

- `left` стає `1`.

- **Ітерація 2:**

- Невідсортована частина: `[1, 4, 2, 5, 3]`. `left=1`, `right=5`.

- **Прохід вправо:** `[1, 2, 4, 3, 5]`. `5` тепер на місці.

- `right` стає `4`.

- **Прохід вліво:** `[1, 2, 3, 4, 5]`. `1` вже був на місці.

- `left` стає `2`.

- **Ітерація 3:**

- Невідсортована частина: `[2, 3, 4]`. `left=2`, `right=4`.

- **Прохід вправо:** обмінів немає. `swapped` залишається `false`.

- Алгоритм завершує роботу.

Аналіз складності

- Часова складність:

- Найгірший випадок: $O(n^2)$. Як і у бульбашкового сортування, якщо масив відсортований у зворотному порядку, кількість порівнянь та обмінів буде квадратичною.

- Середній випадок: $O(n^2)$. Для випадкових даних продуктивність не сильно відрізняється від бульбашкового сортування.

- Найкращий випадок: O(n). Якщо масив вже відсортований, алгоритм виконає одну ітерацію (прохід вправо та назад), не зробить жодного обміну і завершиться.

- Просторова складність: O(1). Алгоритм є “in-place” і не потребує додаткової пам’яті, пропорційної розміру вхідних даних.

Властивості алгоритму

- Стійкість: Алгоритм є стійким, якщо в умові порівняння використовувати строгу нерівність (>). У цьому випадку відносний порядок однакових елементів не зміниться.

- Адаптивність: Алгоритм є адаптивним, оскільки може завершитися достроково, якщо масив стає відсортованим.

Область застосування

Шейкерне сортування, як і бульбашкове, є здебільшого академічним алгоритмом. Його продуктивність $O(n^2)$ робить його непридатним для великих наборів даних.

- Освітні цілі: Це чудовий приклад того, як можна оптимізувати простий алгоритм, вирішивши одну з його ключових проблем (проблему "черепах").

- Невеликі набори даних: Для дуже малих масивів може бути цілком прийнятним через простоту реалізації.

- Специфічні випадки: Може бути трохи ефективнішим за бульбашкове сортування, якщо відомо, що масив містить малі елементи в кінці.

Приклади застосування

Найчастіше цей алгоритм використовується у навчальних програмах та візуалізаторах сортувань, щоб продемонструвати різницю між однонаправленим та двонаправленим підходами. У реальних промислових додатках він практично не зустрічається, поступаючись місцем таким алгоритмам як QuickSort, MergeSort або Timsort.

Переваги та недоліки

Переваги:

- Трохи ефективніше за бульбашкове сортування: Вирішує проблему "черепах", що може дати невеликий приріст продуктивності на певних наборах даних.

- Простота реалізації: Логіка є простим розширенням бульбашкового сортування.

- Стійкість та адаптивність.

- Просторова складність O(1).

Недоліки:

- Низька ефективність: Загальна складність $O(n^2)$ залишається головним недоліком.

- Незначний виграш: Покращення порівняно з оптимізованим бульбашковим сортуванням є незначним і не змінює асимптотичної складності.

- Непрактичність для великих `n`: Повністю поступається алгоритмам з складністю $O(n \log n)$.

Контрольні запитання

1. У чому полягає основна відмінність шейкерного сортування від бульбашкового?

2. Що таке проблема “черепах” у бульбашковому сортуванні, і як шейкерне сортування її вирішує?

3. Поясніть, як працює оптимізація зі звуженням меж **`left`** та **`right`** у шейкерному сортуванні.

4. Чи зміниться асимптотична складність шейкерного сортування порівняно з бульбашковим? Чому?

5. Чи є шейкерне сортування стійким? Обґрунтуйте свою відповідь.

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та порівняльний аналіз шейкерного та бульбашкового сортувань.

2. Мета:

- Реалізувати алгоритм шейкерного сортування.

- Порівняти його продуктивність з класичним оптимізованим бульбашковим сортуванням.

- Дослідити, на яких типах даних шейкерне сортування має переваги.

3. Завдання:

- Реалізувати функцію шейкерного сортування з оптимізацією меж.

- Реалізувати оптимізоване бульбашкове сортування (з прапорцем `swapped`).

- Створити функцію для генерації масиву, що містить “черепаху” (наприклад, `[2, 3, 4, ..., n, 1]`).

- Виміряти час виконання обох алгоритмів на випадкових даних та на даних з “черепахою”.

4. Опис: Лабораторна робота спрямована на практичне порівняння двох дуже близьких алгоритмів. Студент повинен реалізувати обидва та

створити спеціальний набір даних, який, згідно з теорією, має виявити перевагу шейкерного сортування. Це дозволить глибше зрозуміти нюанси їхньої роботи.

5. Порядок виконання:

- Створити проект та реалізувати функції `CocktailShakerSort` та `OptimizedBubbleSort`.

- Реалізувати генератори масивів: випадкового та “черепахового”.

- У головній програмі провести серію тестів на масивах різного розміру.

- Для кожного розміру та типу даних виміряти час виконання обох алгоритмів.

- Результати звести в таблицю.

- Проаналізувати, чи дійсно шейкерне сортування швидше на масиві з “черепахою”.

6. Результат:

- Програма, що реалізує та порівнює два алгоритми.

- Звіт з таблицями результатів та висновками щодо практичної різниці в продуктивності між бульбашковим та шейкерним сортуваннями.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:

- Функція шейкерного сортування.

- Функція бульбашкового сортування.

- Продемонструвати роботу додатку:

- Запуск обох алгоритмів на масиві з “черепахою” для демонстрації різниці в часі.

- Запуск на випадковому масиві.

- Бути готовим відповісти на питання:

- Яка складність обох алгоритмів?

- У чому полягає проблема “черепах” і “зайців”?

- Чому шейкерне сортування не стало значним проривом?

Тести

1. Яку проблему бульбашкового сортування вирішує шейкерне сортування?

1) Велику кількість порівнянь.

2) Нестійкість.

3) Повільне переміщення малих елементів на початок масиву (“черепахи”).

4) Просторову складність O(n).

2. Яка часова складність у найгіршому випадку для шейкерного сортування?

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) $O(n^3)$

3. Після першого повного проходу зліва направо в шейкерному сортуванні, що можна стверджувати напевно?

1) Найменший елемент знаходиться на своїй позиції.

2) Найбільший елемент знаходиться на своїй позиції.

3) Масив повністю відсортований.

4) Половина масиву відсортована.

4. Для якого з наведених масивів шейкерне сортування покаже найбільшу перевагу над бульбашковим?

1) `[1, 2, 3, 4, 5]`

2) `[5, 4, 3, 2, 1]`

3) `[2, 3, 4, 5, 1]`

4) `[3, 1, 4, 2, 5]`

5. Скільки напрямків руху використовує одна повна ітерація шейкерного сортування?

1) Один: зліва направо.

2) Один: справа наліво.

3) Два: зліва направо, а потім справа наліво.

4) Чотири: вправо, вліво, вправо, вліво.

Відповіді:

1. c)

2. c)

3. b)

4. c)

5. c)

Тести підвищеної складності

1. Масив `[3, 8, 2, 9, 5, 1, 6]` сортується методом шейкерного сортування. Яким буде стан масиву після першої повної ітерації (проходу вправо та вліво)?

1) `[1, 3, 2, 8, 5, 6, 9]`

2) `[1, 8, 2, 9, 5, 3, 6]`

3) `[3, 1, 2, 5, 6, 8, 9]`

4) `[1, 3, 8, 2, 9, 5, 6]`

2. Оптимізація шейкерного сортування, що запам'ятовує індекс останнього обміну, найбільше впливає на:

1) Найгірший випадок.

2) Найкращий випадок.

3) Випадки, коли відсортовані частини з'являються на початку та в кінці масиву.

4) Вимоги до пам'яті.

3. Чим шейкерне сортування схоже на сортування гнома?

1) Обидва мають складність O(n) у найкращому випадку.

2) Обидва рухаються в обох напрямках.

3) Обидва є стійкими.

4) Обидва використовують рекурсію.

4. Якщо масив складається з `n-1` однакових елементів і одного унікального, яка складність буде у шейкерного сортування?

1) O(1)

2) O(n)

3) O(n log n)

4) $O(n^2)$

5. Розглянемо масив `A`. Якщо після проходу зліва направо не було жодного обміну, що це означає?

1) Масив був відсортований у зворотному порядку.

2) Масив вже повністю відсортований.

3) Нічого певного, потрібно зробити прохід назад.

4) Найбільший елемент був на першій позиції.

6. Яка з наведених властивостей є спільною для бульбашкового, шейкерного та сортування вставкою?

1) Усі вони мають складність O(n) в найкращому випадку.

2) Усі вони нестабільні.

3) Усі вони є “онлайн” алгоритмами.

4) Усі вони мають просторову складність O(1).

7. На масиві `[1, 2, 3, 4, 5]`, скільки всього порівнянь зробить оптимізоване шейкерне сортування?

1) 4

2) 8

3) 10

4) 0

8. Порівняно з сортуванням Шелла, шейкерне сортування:

1) Має кращу асимптотичну складність.

2) Є значно простішим для реалізації.

3) Є стійким, на відміну від сортування Шелла.

4) Використовує більше пам’яті.

9. Яка з оптимізацій НЕ застосовується до шейкерного сортування?

1) Дострокове завершення, якщо не було обмінів.

2) Звуження робочого діапазону масиву після кожного проходу.

3) Використання бінарного пошуку для знаходження місця елемента.

4) Відстеження індексу останнього обміну для звуження меж.

10. Шейкерне сортування також називають “сортуванням перемішуванням”. Чому?

1) Бо воно випадковим чином переставляє елементи.

2) Через аналогію з рухом рідини в шейкері туди-сюди.

3) Бо воно “перемішує” відсортовану та невідсортовану частини.

4) Назва є історичною і не має логічного пояснення.

Відповіді:

1. 1) `[1, 3, 2, 8, 5, 6, 9]`. Прохід вправо дає `[3, 2, 8, 5, 1, 6, 9]`. Прохід вліво з цього стану дає `[1, 3, 2, 8, 5, 6, 9]`.

2. 3) Випадки, коли відсортовані частини з’являються на початку та в кінці масиву. Ця оптимізація дозволяє не сканувати ті частини, які вже гарантовано впорядковані.

3. 2) Обидва рухаються в обох напрямках. Сортування гнома рухається вправо, поки елементи впорядковані, і повертається вліво, коли знаходить порушення порядку.

4. 2) O(n). Алгоритм швидко поставить унікальний елемент на місце (за одну-дві ітерації), а потім зробить ще один прохід без обмінів, щоб переконатися, що все відсортовано.

5. 2) Масив вже повністю відсортований. Якщо прохід зліва направо не зробив жодного обміну, це означає, що для всіх `i`, `A[i] <= A[i+1]`, що є визначенням відсортованого масиву.

6. 4) Усі вони мають просторову складність O(1). Це їхня спільна риса як простих “in-place” сортувань.

7. 1) 4. Алгоритм зробить один прохід зліва направо, виконає `n-1 = 4` порівняння, не знайде обмінів і завершить роботу.

8. 3) Є стійким, на відміну від сортування Шелла. Це одна з небагатьох переваг простих сортувань над більш просунутими.

9. 3) Використання бінарного пошуку для знаходження місця елемента. Ця техніка застосовна до сортування вставкою, але не має сенсу для сортувань, що базуються на обміні сусідніх елементів.

10. 2) Через аналогію з рухом рідини в шейкері туди-сюди. Назва візуально описує двонаправлений рух елементів.

Завдання для самостійної роботи

1. Реалізуйте шейкерне сортування з найповнішою оптимізацією (відстеженням індексів останніх обмінів).

2. Напишіть програму, яка візуалізує роботу шейкерного сортування, виділяючи кольором межі `left` та `right`.

3. Порівняйте на практиці продуктивність вашої реалізації з сортуванням гнома (Gnome Sort).

4. Реалізуйте рекурсивну версію шейкерного сортування.

5. Адаптуйте алгоритм для сортування двобічно зв'язного списку. Чи буде це ефективніше, ніж для масиву?

6. Дослідіть, як наявність великої кількості дублікатів впливає на кількість обмінів та порівнянь.

7. Створіть "антитест" для шейкерного сортування — масив, на якому воно працюватиме максимально довго.

8. Напишіть тести, які перевіряють стійкість вашої реалізації.

9. Модифікуйте алгоритм так, щоб він сортував масив за спаданням.

10. Порівняйте кількість операцій обміну для шейкерного сортування та сортування вибором на різних типах даних.

Теми лабораторних робіт

1. Порівняльний аналіз продуктивності `Bubble Sort`, `Cocktail Shaker Sort` та `Gnome Sort`.

2. Розробка інтерактивного візуалізатора, що показує різницю в поведінці "черепах" і "зайців" у бульбашковому та шейкерному сортуваннях.

3. Емпіричне дослідження впливу різних оптимізацій на продуктивність шейкерного сортування.

4. Застосування шейкерного сортування для сортування даних, що надходять у двосторонню чергу (deque).

5. Створення гібридного алгоритму: використання `QuickSort` з переходом на `Cocktail Shaker Sort` для малих підмасивів.

6. Аналіз кеш-ефективності шейкерного сортування порівняно з іншими простими сортуваннями.

7. Реалізація паралельної версії шейкерного сортування, де проходи вправо і вліво виконуються в різних потоках.

8. Дослідження поведінки шейкерного сортування на масивах, що мають специфічні патерни (наприклад, відсортовані по синусоїді).

9. Оцінка продуктивності при сортуванні масивів об'єктів з дорогими операціями порівняння.

10. Розробка “універсального” простого сортувальника, який динамічно обирає між `Insertion Sort` та `Cocktail Sort` залежно від характеристик даних.

9.1.6 Сортування гнома

Теоретичні відомості

Сортування гнома — це простий та елегантний алгоритм сортування, який концептуально схожий на сортування вставкою, але відрізняється надзвичайно простою реалізацією без вкладених циклів. Свою назву алгоритм отримав завдяки аналогії, запропонованій Діком Грюне: уявіть садового гнома, який сортує лінію квіткових горщиків.

Основний принцип роботи (аналогія з гномом):

1. Гном стоїть біля одного з горщиків. Він дивиться на горщик перед собою та на горщик позаду себе.

2. Якщо вони стоять у правильному порядку (попередній менший або рівний за поточний), гном робить крок вперед до наступного горщика.

3. Якщо вони стоять у неправильному порядку (попередній більший за поточний), гном міняє їх місцями і робить крок **назад**, щоб перевірити, чи не порушив цей обмін порядок попередніх горщиків.

4. Якщо гном доходить до самого початку, він не може зробити крок назад, тому просто робить крок уперед.

5. Коли гном доходить до кінця ряду горщиків, робота вважається завершеною.

По суті, гном рухається вперед, поки масив локально відсортований. Знайшовши порушення порядку, він виправляє його і повертається назад, несучи елемент-“порушник” на його правильну позицію, подібно до того, як сортування вставкою зсуває елементи.

Порівняння з сортуванням вставкою:

Сортування гнома виконує ту ж саму послідовність обмінів, що і сортування вставкою. Однак, якщо сортування вставкою використовує вкладений цикл для пошуку місця вставки, сортування гнома досягає того ж результату за допомогою одного циклу, керуючи лише позицією “гнома” (поточного індексу).

Псевдокод алгоритму

Реалізація сортування гнома вражаюче коротка.

```
function GnomeSort(array A)
  n = length(A)
  index = 0

  while index < n
    // Якщо ми на початку або поточний елемент не порушує порядок, ру
хаємося вперед.
    if index == 0 or A[index] >= A[index-1]
      index = index + 1
    // Якщо порядок порушено, міняємо елементи місцями і рухаємося на
зад.
    else
      swap(A[index], A[index-1])
      index = index - 1
    end if
  end while
end function
```

- `A`: Масив для сортування.

- `n`: Розмір масиву.

- `index`: Поточна позиція “гнома”. Ця змінна є єдиним інструментом керування потоком алгоритму.

Приклад роботи

Розглянемо сортування масиву `A = [3, 4, 2, 5, 1]`.

- `index = 0`: `index == 0`, `index` стає `1`. Масив: `[3, 4, 2, 5, 1]`.

- `index = 1`: `A[1](4) >= A[0](3)`. Порядок правильний. `index` стає `2`. Масив: `[3, 4, 2, 5, 1]`.

- `index = 2`: `A[2](2) < A[1](4)`. **Порушення!**

- `swap(A[2], A[1])`. Масив: `[3, 2, 4, 5, 1]`.

- `index` стає `1`.

- `index = 1`: `A[1](2) < A[0](3)`. **Порушення!**

- `swap(A[1], A[0])`. Масив: `[2, 3, 4, 5, 1]`.

- `index` стає `0`.

- `index = 0`: `index == 0`, `index` стає `1`. Масив: `[2, 3, 4, 5, 1]`.

- `index = 1`: `A[1](3) >= A[0](2)`. `index` стає `2`.

- `index = 2`: `A[2](4) >= A[1](3)`. `index` стає `3`.

- `index = 3`: `A[3](5) >= A[2](4)`. `index` стає `4`.

- `index = 4`: `A[4](1) < A[3](5)`. **Порушення!**

- `swap(A[4], A[3])`. Масив: `[2, 3, 4, 1, 5]`.

- `index` стає `3`.

- `index = 3`: `A[3](1) < A[2](4)`. **Порушення!**

- `swap(A[3], A[2])`. Масив: `[2, 3, 1, 4, 5]`.

- `index` стає `2`.

- `index = 2`: `A[2](1) < A[1](3)`. **Порушення!**

- `swap(A[2], A[1])`. Масив: `[2, 1, 3, 4, 5]`.

- `index` стає `1`.

- `index = 1`: `A[1](1) < A[0](2)`. **Порушення!**

- `swap(A[1], A[0])`. Масив: `[1, 2, 3, 4, 5]`.

- `index` стає `0`.

- `index = 0`: `index` стає `1`.

- `index = 1, 2, 3, 4`: Порядок правильний, `index` послідовно збільшується до `5`.

- `index = 5`: `index < n` (5 < 5) є `false`. Цикл завершується.

Аналіз складності

- Часова складність:

- Найгірший випадок: $O(n^2)$. Виникає на зворотньо відсортованому масиві. Гном змушений буде “тягнути” кожен елемент через увесь відсортований підмасив.

- Середній випадок: $O(n^2)$.

- Найкращий випадок: O(n). Якщо масив уже відсортований, гном просто пройде по ньому з початку в кінець за `n` кроків, не зробивши жодного обміну.

- Просторова складність: O(1). Алгоритм є “in-place” і потребує лише одну додаткову змінну для індексу.

Властивості алгоритму

- Стійкість: Алгоритм є стійким. Умова порівняння `A[index] >= A[index-1]` гарантує, що елементи з однаковими значеннями не будуть мінятися місцями.

- Адаптивність: Алгоритм є адаптивним, оскільки працює за лінійний час на вже відсортованих або майже відсортованих даних.

- Надзвичайна простота: Це одна з ключових переваг. Його можна написати буквально в кілька рядків коду, що робить його привабливим для швидкого прототипування.

Область застосування

Як і інші прості сортування, сортування гнома не підходить для великих обсягів даних через свою квадратичну складність.

- Освітні цілі: Це, мабуть, найпростіший для розуміння та реалізації алгоритм сортування, що робить його ідеальним для знайомства з концепцією сортування.

- Прототипування та швидкі скрипти: Коли потрібно швидко написати сортування для невеликого набору даних і немає часу чи бажання згадувати більш складні алгоритми.

- “Code golf”: У змаганнях, де метою є написання найкоротшого коду для вирішення задачі.

- Сортування майже відсортованих даних: Завдяки адаптивності, може бути ефективним, якщо дані вже частково впорядковані.

Приклади застосування

Студент на олімпіаді з програмування вирішує задачу, де потрібно відсортувати невелику кількість елементів (наприклад, до 50). Замість того, щоб витрачати час на написання QuickSort і ризикувати помилитися, він може за хвилину написати безпомилкове сортування гнома і бути впевненим у його коректності.

Переваги та недоліки

Переваги:

- Екстремальна простота реалізації: найпростіший для написання серед усіх сортувань.

- Мінімалістичність коду: не має вкладених циклів.

- Стійкість та адаптивність.

- Просторова складність O(1).

Недоліки:

- Низька ефективність: складність $O(n^2)$ робить його дуже повільним для великих або випадкових масивів.

- Виконує ту ж роботу, що й сортування вставкою, але може бути менш інтуїтивним для аналізу через нелінійний рух індексу.

Контрольні запитання

1. Поясніть аналогію з садовим гномом, що сортує горщики. Яким діям гнома відповідають кроки алгоритму?

2. Чому сортування гнома вважається схожим на сортування вставкою? У чому їхня принципова відмінність у реалізації?

3. За яких умов “гном” (поточний індекс) рухається вперед, а за яких — назад?

4. Наведіть приклад масиву з 4 елементів, на якому гном буде рухатися тільки вперед.

5. Наведіть приклад масиву з 4 елементів, на якому гном буде змушений повернутися до самого початку.

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз сортування гнома.

2. Мета:

- Реалізувати алгоритм сортування гнома.

- Порівняти його реалізацію та продуктивність з сортуванням вставкою.

- Візуалізувати рух індексу в алгоритмі.

3. Завдання:

- Реалізувати функцію сортування гнома.

- Реалізувати функцію сортування вставкою.

- Додати в обидві функції лічильники порівнянь та обмінів/присвоєнь.

- Провести експерименти на однакових наборах даних (випадкових, відсортованих, зворотних) та порівняти кількість операцій.

- Написати функцію, яка для невеликого масиву виводить на кожному кроці сортування гнома поточний індекс та стан масиву.

4. Опис: Лабораторна робота має на меті показати, що два різні за структурою коду алгоритми (Гнома та Вставкою) можуть бути функціонально еквівалентними за результатом і кількістю операцій. Візуалізація руху індексу допоможе краще зрозуміти незвичну логіку сортування гнома.

5. Порядок виконання:

- Створити проект та реалізувати функції `GnomeSort` та `InsertionSort` з лічильниками.

- Реалізувати генератори масивів.

- Реалізувати функцію візуалізації для `GnomeSort`.

- Запустити візуалізацію на масиві з 5-7 елементів.

- Провести заміри кількості операцій для обох алгоритмів на масивах різного розміру та типу.

- Порівняти результати та зробити висновок про еквівалентність їхньої роботи.

6. Результат:

- Програма, що реалізує обидва алгоритми та дозволяє порівняти їхню внутрішню роботу.

- Звіт з таблицями, що підтверджують, що кількість обмінів для обох алгоритмів є однаковою, та висновками щодо простоти реалізації.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
    - Функція сортування гнома.
    - Функція візуалізації руху індексу.
- Продемонструвати роботу додатку:
    - Покрокове виконання сортування гнома на невеликому масиві.
    - Порівняння лічильників операцій для сортування гнома та вставкою на однакових даних.
- Бути готовим відповісти на питання:
    - Яка складність алгоритму і чому?
    - Чому цей алгоритм не має вкладених циклів?
    - Який реальний алгоритм є аналогом сортування гнома?

Тести

1. Яка головна перевага сортування гнома?

1) Найкраща швидкість серед усіх алгоритмів.

2) Простота реалізації та мінімалізм коду.

3) Здатність сортувати рядки краще за числа.

4) Просторова складність O(log n).

2. Поведінка якого алгоритму найбільш схожа на сортування гнома?

1) Сортування вибором.

2) Сортування вставкою.

3) Сортування злиттям.

4) Бульбашкове сортування.

3. Що робить “гном” (поточний індекс), коли знаходить два елементи в неправильному порядку (`A[i] < A[i-1]`)?

1) Ігнорує їх і рухається вперед.

2) Зупиняє алгоритм.

3) Міняє їх місцями і рухається вперед.

4) Міняє їх місцями і рухається назад.

4. Яка часова складність у найкращому випадку для сортування гнома?

1) O(1)

2) O(n)

3) O(n log n)

4) $O(n^2)$

5. Для сортування масиву `[2, 4, 1, 3]` скільки разів “гном” (індекс) буде змушений рухатися назад?

1) 1

2) 2

3) 3

4) 4

Відповіді:

1. b)

2. b)

3. d)

4. b)

5. c)

Тести підвищеної складності

1. Розглянемо масив `[4, 2, 5, 1, 3]`. Який максимальний індекс, якого досягне “гном”, перш ніж вперше повернутися до індексу 0?

1) 1

2) 2

3) 3

4) 4

2. Якщо в реалізації сортування гнома замінити умову `A[index] >= A[index-1]` на `A[index] > A[index-1]`, що станеться?

1) Алгоритм перестане працювати.

2) Алгоритм стане нестійким.

3) Нічого не зміниться.

4) Алгоритм стане швидшим.

3. Яку кількість обмінів виконає сортування гнома на масиві `[3, 2, 1]`?

1) 1

2) 2

3) 3

4) 4

4. Чим сортування гнома відрізняється від шейкерного сортування?

1) Гном рухається назад лише за потреби, а шейкер — на кожній ітерації.

2) Гном є стійким, а шейкер — ні.

3) Гном має складність O(n), а шейкер $O(n^2)$.

4) Гном використовує більше пам’яті.

5. Якщо “гном” знаходиться на позиції `k` і тільки що виконав обмін, яка буде його наступна позиція?

1) `k+1`

2) `k`

3) `k-1`

4) `0`

6. Сортування гнома іноді порівнюють з бульбашковим сортуванням. Яка дія в сортуванні гнома аналогічна “спливанню” елемента в бульбашковому сортуванні?

1) Послідовний рух гнома вперед.

2) Один крок гнома назад.

3) Серія кроків гнома назад, що переносить один елемент на його позицію.

4) Перший крок гнома з позиції 0 на 1.

7. На масиві `[1, 3, 2, 4]` скільки разів виконається операція `index = index + 1`?

1) 4

2) 5

3) 6

4) 7

8. Якщо алгоритм сортування гнома застосувати до двобічно зв’язного списку, чи зміниться його асимптотична складність?

1) Так, покращиться до O(n log n).

2) Так, погіршиться до $O(n^3)$.

3) Ні, залишиться такою ж, оскільки операції переходу та обміну є ефективними.

4) Алгоритм неможливо застосувати до зв’язного списку.

9. Що станеться, якщо ініціалізувати `index = 1` замість `index = 0`?

1) Нічого, алгоритм працюватиме так само.

2) Алгоритм пропустить порівняння першого елемента і може дати невірний результат.

3) Це стандартна реалізація.

4) Алгоритм зациклиться.

10. Яке твердження є хибним щодо сортування гнома?

1) Це адаптивний алгоритм сортування.

2) Це найпростіший для реалізації алгоритм сортування.

3) Він виконує таку ж кількість порівнянь, як і сортування вставкою.

4) Він є стійким.

Відповіді:

1. 3) 3. `i=1`(`4>A[0]`) -> `i=2`(`5>4`) -> `i=3`. На `i=3`(`1<5`) відбувається перший обмін і рух назад.

2. 2) Алгоритм стане нестійким. Умова `>` призведе до обміну однакових елементів, порушуючи їхній відносний порядок.

3. 3) 3. `[3,2,1]`->`[2,3,1]`->`[2,1,3]`->`[1,2,3]`. Всього 3 обміни.

4. 1) Гном рухається назад лише за потреби, а шейкер — на кожній ітерації. Це ключова відмінність у їхній логіці руху.

5. 3) `k-1`. Після обміну гном завжди робить крок назад, щоб перевірити результат обміну.

6. 3) Серія кроків гнома назад, що переносить один елемент на його позицію. Цей процес аналогічний внутрішньому циклу сортування вставкою, який ставить елемент на місце.

7. 2) 5. Трасування: `i=0->1`, `1->2`, `2->3`(`2<3`), `3->2`, `2->3`, `3->4`, `4->5`(вихід). Разом 5.

8. 3) Ні, залишиться такою ж, оскільки операції переходу та обміну є ефективними. Перехід до попереднього/наступного елемента та обмін даних у вузлах є операціями O(1).

9. 1) Нічого, алгоритм працюватиме так само. Перша ітерація циклу `while` з `index=1` перевірить `A[1] >= A[0]` і зробить те саме, що і версія з `index=0`.

10. 3) Він виконує таку ж кількість порівнянь, як і сортування вставкою. Це не завжди вірно. Кількість обмінів однакова, але кількість порівнянь може відрізнятися через різну логіку керування.

Завдання для самостійної роботи

1. Реалізуйте сортування гнома, використовуючи цикл `for` замість `while` (може знадобитися змінювати лічильник циклу).

2. Напишіть програму, яка візуалізує “шлях гнома” по масиву, малюючи графік залежності індексу від часу.

3. Порівняйте на практиці кількість операцій порівняння для сортування гнома та сортування вставкою. Перевірте, чи справді вони відрізняються.

4. Реалізуйте “оптимізоване” сортування гнома, яке запам’ятовує, де востаннє був порядок, і може “стрибати” вперед. Проаналізуйте його ефективність.

5. Адаптуйте алгоритм для сортування масиву рядків за довжиною.

6. Напишіть тести, які перевіряють стійкість вашої реалізації.

7. Створіть масив, на якому сортування гнома виконає максимальну кількість кроків назад.

8. Дослідіть, як можна було б розпаралелити сортування гнома. Чи це взагалі можливо?

9. Порівняйте простоту коду сортування гнома, вставкою та бульбашкового. Оцініть суб’єктивно, який з них найлегше написати з пам’яті.

10. Модифікуйте алгоритм, щоб він сортував у зворотному порядку.

Теми лабораторних робіт

1. Функціональна еквівалентність `Gnome Sort` та `Insertion Sort`: порівняльний аналіз кількості операцій.

2. Розробка інтерактивного візуалізатора, що демонструє рух “гнома” по масиву.

3. Порівняльний аналіз “руху назад” у `Gnome Sort` та `Cocktail Shaker Sort`.

4. Застосування сортування гнома для сортування списків у функціональних мовах програмування.

5. Аналіз продуктивності `Gnome Sort` на майже відсортованих даних порівняно з `Timsort`.

6. Створення “Code Golf” змагання: реалізація `Gnome Sort` на різних мовах програмування з найменшою кількістю символів.

7. Дослідження можливості застосування сортування гнома для сортування потокових даних.

8. Аналіз поведінки сортування гнома на даних зі специфічними патернами (наприклад, масив, де відсортовані перша та друга половини).

9. Оцінка практичної продуктивності `Gnome Sort` як фінального етапу для гібридних алгоритмів.

10. Історія та походження “дурних” сортувань: від `Bogosort` до `Gnome Sort`.

9.2. Ефективні сортування: вихід за межі квадратичної складності

Переходячи від простих сортувань, ми вступаємо в еру справжньої ефективності. Якщо алгоритми на кшталт бульбашкового чи сортування вибором є чудовими для навчання, їхня часова складність $O(n^2)$ стає непереборною перешкодою при роботі з великими наборами даних. Сортування мільйона елементів таким методом може зайняти години, тоді як ефективні алгоритми впораються за лічені секунди.

У цьому розділі ми дослідимо “велику трійку” алгоритмів сортування, які визначають стандарти продуктивності в сучасній інформатиці та працюють із середньою часовою складністю $O(n\log n)$ — теоретично найкращим можливим результатом для сортувань, що базуються на порівняннях.

- **Сортування злиттям (Merge Sort):** Класичний приклад стратегії “розділяй і володарюй”. Цей алгоритм рекурсивно ділить масив навпіл, сортує ці половини, а потім ефективно зливає їх разом. Його головні козирі — це гарантована продуктивність $O(n\log n)$ у будь-якому випадку та стійкість, що робить його незамінним у багатьох системних задачах.

- **Швидке сортування (Quick Sort):** Часто найшвидший на практиці, цей алгоритм також використовує підхід “розділяй і володарюй”, але робить це інакше. Він обирає “опорний елемент” (pivot) і розділяє масив на дві частини: елементи, менші за опорний, та елементи, більші за нього. Ми розглянемо декілька ключових стратегій розділення, кожна з яких має свої нюанси та переваги: класичну схему Хоара, просту для розуміння схему Ломуто та сучасну оптимізацію Dual-Pivot, яка використовується в стандартних бібліотеках Java.

- **Пірамідальне сортування (Heap Sort):** Унікальний та елегантний алгоритм, що використовує структуру даних “піраміда” (або бінарна купа) для досягнення складності $O(n\log n)$ прямо “на місці” (in-place), не потребуючи додаткової пам’яті. Він перетворює вхідний масив на піраміду, а потім послідовно витягує з неї найбільший елемент, будуючи відсортовану послідовність.

Знайомство з цими алгоритмами — це не просто вивчення нових методів. Це розуміння фундаментальних принципів, які лежать в основі швидких та масштабованих обчислень у всьому, від баз даних до машинного навчання.

9.2.1 Сортування злиттям

Теоретичні відомості

Сортування злиттям (Merge Sort)— це один з найефективніших та найнадійніших алгоритмів сортування, який є взірцевим прикладом парадигми “Розділяй і володарюй”. На відміну від QuickSort, його продуктивність не залежить від початкових даних, що гарантує стабільний час роботи. Алгоритм був винайдений Джоном фон Нейманом у 1945 році.

Основний принцип роботи “Розділяй і володарюй”:

Стратегія алгоритму складається з трьох кроків, що застосовуються рекурсивно:

1. **Розділяй:** Масив (або його частина), що складається з `n` елементів, ділиться на дві рівні (або майже рівні) частини, кожна розміром `n/2`. Цей крок є тривіальним і полягає лише в обчисленні середнього індексу.

2. **Володарюй:** Кожна з двох отриманих частин сортується рекурсивно, викликаючи для неї ту ж саму функцію сортування злиттям. Базовим випадком рекурсії є підмасив розміром 1, який за визначенням вже є відсортованим.

3. **Комбінуй:** Два відсортовані підмасиви зливаються в один єдиний відсортований масив. Цей крок є ключовим і найскладнішим в алгоритмі.

Процедура злиття

Процедура `merge` приймає на вхід три індекси `left`, `mid`, `right`, які визначають два сусідні відсортовані підмасиви: `A[left...mid]` та `A[mid+1...right]`. Її завдання — злити їх в один відсортований підмасив `A[left...right]`.

Алгоритм процедури `злиття`:

1. Створити два тимчасові масиви, `L` та `R`, і скопіювати в них дані з `A[left...mid]` та `A[mid+1...right]` відповідно.

2. Ініціалізувати три вказівники (індекси): `i` для масиву `L`, `j` для масиву `R` (обидва на початку), та `k` для основного масиву `A` (починаючи з `left`).

3. У циклі порівнювати `L[i]` та `R[j]`. Менший з двох елементів копіювати в `A[k]` і збільшувати відповідний вказівник (`i` або `j`) та вказівник `k`.

4. Коли один з тимчасових масивів (`L` або `R`) закінчиться, скопіювати залишок іншого масиву в кінець масиву `A`.

Псевдокод процедури

```
// A - основний масив, left, mid, right - індекси
function merge(array A, int left, int mid, int right)
  // 1. Створюємо тимчасові масиви
  n1 = mid - left + 1
  n2 = right - mid
  array L[n1], R[n2]

  // 2. Копіюємо дані в тимчасові масиви
  for i from 0 to n1-1
    L[i] = A[left + i]
  end for
  for j from 0 to n2-1
    R[j] = A[mid + 1 + j]
  end for

  // 3. Зливаємо тимчасові масиви назад в A[left...right]
  i = 0 // Початковий індекс першого підмасиву (L)
  j = 0 // Початковий індекс другого підмасиву (R)
  k = left // Початковий індекс злитого підмасиву (в A)

  while i < n1 and j < n2
    if L[i] <= R[j]
      A[k] = L[i]
      i = i + 1
    else
      A[k] = R[j]
      j = j + 1
    end if
    k = k + 1
  end while

  // 4. Копіюємо залишки, якщо вони є
  while i < n1
    A[k] = L[i]
    i = i + 1
    k = k + 1
  end while
  while j < n2
```

```
    A[k] = R[j]
    j = j + 1
    k = k + 1
  end while
end function
```

Рекурсивний підхід

Це класична реалізація сортування злиттям.

Псевдокод рекурсивного Merge Sort

```
function MergeSort(array A, int left, int right)
  if left < right
    // 1. Розділяй: знаходимо середину
    mid = left + (right - left) / 2 // Щоб уникнути переповнення для
великих left, right

    // 2. Володарюй: рекурсивно сортуємо дві половини
    MergeSort(A, left, mid)
    MergeSort(A, mid + 1, right)

    // 3. Комбінуй: зливаємо відсортовані половини
    merge(A, left, mid, right)
  end if
end function
```

Приклад роботи (рекурсивний підхід)

Розглянемо сортування масиву `A = [38, 27, 43, 3, 9, 82, 10]`.

1. `MergeSort([38, 27, 43, 3, 9, 82, 10])`
2. Розділяє на `[38, 27, 43, 3]` і `[9, 82, 10]`.
3. `MergeSort([38, 27, 43, 3])` -> розділяє на `[38, 27]` і `[43, 3]`.
4. `MergeSort([38, 27])` -> розділяє на `[38]` і `[27]`. Вони відсортовані.
5. `merge([38], [27])` -> повертає `[27, 38]`.
6. `MergeSort([43, 3])` -> розділяє на `[43]` і `[3]`.
7. `merge([43], [3])` -> повертає `[3, 43]`.
8. `merge([27, 38], [3, 43])` -> повертає `[3, 27, 38, 43]`. Ліва частина відсортована.
9. Аналогічно для правої частини `[9, 82, 10]`, яка після сортування стане `[9, 10, 82]`.
10. Фінальне злиття: `merge([3, 27, 38, 43], [9, 10, 82])` -> `[3, 9, 10, 27, 38, 43, 82]`.

Ітеративний підхід

Альтернативна реалізація, яка дозволяє уникнути рекурсії та пов'язаних з нею накладних витрат. Ідея полягає в тому, щоб зливати підмасиви знизу вгору.

1. Спочатку зливаються підмасиви розміром 1, щоб утворити відсортовані підмасиви розміром 2.

2. Потім зливаються підмасиви розміром 2, щоб утворити відсортовані підмасиви розміром 4.

3. Процес триває, подвоюючи розмір підмасивів на кожному кроці, доки весь масив не буде відсортовано.

Псевдокод ітеративного Merge Sort

```
function IterativeMergeSort(array A)
  n = length(A)
  // size - поточний розмір підмасивів для злиття
  for size from 1 to n-1 step size*2
    // left_start - початковий індекс лівого підмасиву
    for left_start from 0 to n-1 step size*2
      mid = min(left_start + size - 1, n - 1)
      right_end = min(left_start + 2*size - 1, n - 1)

      // Викликаємо процедуру злиття для A[left_start...mid] та A[mid
+1...right_end]
      merge(A, left_start, mid, right_end)
    end for
  end for
end function
```

Аналіз складності

- Часова складність: O(n log n).

- Це справедливо для найкращого, середнього та найгіршого випадків.

- Пояснення: рекурсивне ділення масиву створює дерево викликів глибиною $O(\log n)$. На кожному рівні цього дерева процедура `merge` обробляє в сумі `n` елементів. Загальна робота: `(кількість рівнів) × (робота на рівень) = O(log n) × O(n) = O(n log n)`.

- Просторова складність: O(n).

- Основний недолік алгоритму. Для процедури `merge` потрібен тимчасовий масив, розмір якого в найгіршому випадку дорівнює розміру вихідного масиву.

Властивості алгоритму

- Стійкість: Алгоритм є стійким. Під час злиття, якщо `L[i] == R[j]`, ми можемо завжди першим брати елемент з лівого масиву `L`. Це гарантує, що відносний порядок однакових елементів не зміниться.

- Не “in-place”: Вимагає додаткової пам’яті.

- Не адаптивний: Виконує однакову кількість операцій незалежно від початкової впорядкованості масиву.

Область застосування

Завдяки своїй стабільності та гарантованій продуктивності, Merge Sort є дуже важливим алгоритмом.

- Зовнішнє сортування: Ідеально підходить для сортування даних, що не вміщуються в оперативну пам’ять. Алгоритм може читати частини даних (блоки) з диска, сортувати їх в пам’яті, а потім ефективно зливати відсортовані блоки.

- Сортування зв’язних списків: На відміну від масивів, для зв’язних списків Merge Sort можна реалізувати з просторовою складністю $O(1)$, оскільки злиття можна виконати, просто змінюючи вказівники, без копіювання у тимчасовий список.

- Коли стійкість є вимогою: У задачах, де важливо зберегти початковий порядок елементів з однаковими ключами (наприклад, сортування таблиці за однією колонкою, не порушуючи порядок, заданий іншою).

- Компонент інших алгоритмів: Використовується в деяких реалізаціях Timsort та в алгоритмах обчислювальної геометрії.

Приклади застосування

Система управління базою даних повинна відсортувати величезну таблицю клієнтів за датою реєстрації. Файл з таблицею має розмір 100 ГБ, а оперативна пам’ять сервера — 16 ГБ. Система може читати файл по 1 ГБ, сортувати цей блок в пам’яті (наприклад, за допомогою QuickSort) і зберігати його як тимчасовий відсортований файл. Після створення 100 таких файлів, система використовує багатопрохідну процедуру злиття (аналогічну `merge` в Merge Sort) для об’єднання їх в один фінальний відсортований файл.

Переваги та недоліки

Переваги:

- Гарантована часова складність O(n log n): Дуже надійний, без “найгірших випадків”, як у QuickSort.

- Стійкість: Зберігає відносний порядок однакових елементів.

- Висока паралелізованість: Рекурсивні виклики для підмасивів є незалежними і можуть виконуватися паралельно.

- Ефективність для зовнішнього сортування.

Недоліки:

- Вимагає додаткової пам’яті O(n): Це головний недолік для сортування великих масивів в оперативній пам’яті.

- Повільніший за QuickSort в середньому: На типових задачах в пам’яті QuickSort зазвичай працює швидше через менші накладні витрати та кращу кеш-ефективність.

Контрольні запитання

1. Опишіть три кроки парадигми “Розділяй і володарюй” на прикладі Merge Sort.

2. Яка роль процедури `merge` і чому вона є ключовою для роботи всього алгоритму?

3. Поясніть, чому часова складність Merge Sort завжди становить $O(n \log n)$.

4. Чому класична реалізація Merge Sort потребує $O(n)$ додаткової пам’яті? Як можна уникнути цього при сортуванні зв’язних списків?

5. Поясніть, чому Merge Sort є стійким алгоритмом. Наведіть приклад.

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та порівняльний аналіз рекурсивної та ітеративної версій сортування злиттям.

2. Мета:

- Реалізувати рекурсивну (top-down) та ітеративну (bottom-up) версії сортування злиттям.

- Порівняти їхню продуктивність та використання пам’яті.

- Реалізувати сортування злиттям для зв’язного списку.

3. Завдання:

- Реалізувати функцію `MergeSort_Recursive`.

- Реалізувати функцію `MergeSort_Iterative`.

- Створити структуру даних для однозв’язного списку та реалізувати для неї сортування злиттям.

- Виміряти час виконання та пікове використання пам’яті для всіх трьох реалізацій на масивах/списках різного розміру.

4. Опис: Лабораторна робота дозволяє глибоко зрозуміти один з найважливіших алгоритмів сортування. Студент повинен реалізувати не тільки класичну рекурсивну версію, але й ітеративну, щоб побачити переваги відмови від рекурсії. Реалізація для зв’язного списку продемонструє, як структура даних може кардинально змінити вимоги алгоритму до пам’яті.

5. Порядок виконання:

- Створити проект та реалізувати процедуру `merge`.

- На її основі реалізувати рекурсивну та ітеративну версії сортування.

- Реалізувати сортування для зв’язного списку (ключова частина — написання функції злиття двох відсортованих списків).

- Написати програму-бенчмарк, яка тестує всі реалізації на даних різного розміру (наприклад, 10 000, 100 000, 1 000 000 елементів).

- Зібрати дані про час та використання пам’яті.

- Проаналізувати результати, звернувши увагу на різницю між рекурсивним та ітеративним підходами, а також на переваги версії для зв’язного списку.

6. Результат:

- Програма з трьома реалізаціями сортування злиттям.

- Звіт з аналізом продуктивності, що показує переваги та недоліки кожного підходу.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
    - Процедура `merge`.
    - Рекурсивна та ітеративна реалізації для масивів.
    - Реалізація для зв’язного списку.
- Продемонструвати роботу додатку:
    - Покроковий процес злиття двох невеликих масивів.
    - Порівняння часу роботи рекурсивної та ітеративної версій.

- Бути готовим відповісти на питання:
    - Яка складність алгоритму за часом та пам’яттю?
    - У чому різниця між top-down та bottom-up підходами?
    - Чому Merge Sort є стійким?
    - Де на практиці застосовується Merge Sort?

Тести

1. Яка парадигма лежить в основі сортування злиттям?

1) Жадібний алгоритм.

2) Динамічне програмування.

3) Розділяй і володарюй.

4) Метод грубої сили (Brute-force).

2. Яка часова складність у найгіршому випадку для сортування злиттям?

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) O(log n)

3. Основний недолік сортування злиттям при роботі з масивами:

1) Нестійкість.

2) Складність $O(n^2)$ у найгіршому випадку.

3) Потреба в додатковій пам’яті O(n).

4) Складність реалізації.

4. Для якої структури даних сортування злиттям може бути реалізоване з просторовою складністю O(1)?

1) Масив.

2) Зв’язний список.

3) Дерево.

4) Хеш-таблиця.

5. Що є “базовим випадком” у рекурсивній реалізації сортування злиттям?

1) Масив з двох елементів.

2) Порожній масив.

3) Масив з одного елемента.

4) Масив, що вже відсортований.

Відповіді:

1. c)
2. b)
3. c)
4. b)
5. c)

Тести підвищеної складності

1. Скільки рівнів рекурсії буде для сортування злиттям масиву з 1000 елементів?

1) 1000

2) 100

3) 10

4) 3

2. Яке твердження про ітеративне (bottom-up) сортування злиттям є істинним?

1) Воно завжди швидше за рекурсивне.

2) Воно уникає накладних витрат, пов’язаних з глибокою рекурсією.

3) Воно працює з просторовою складністю O(1) для масивів.

4) Воно є нестійким.

3. Чому сортування злиттям є добрим вибором для зовнішнього сортування?

1) Бо воно не використовує пам’ять.

2) Бо його процедура злиття може послідовно обробляти дані з файлів, не завантажуючи їх повністю в пам’ять.

3) Бо воно найшвидше працює з даними на жорсткому диску.

4) Бо воно не потребує порівнянь.

4. При злитті двох відсортованих підмасивів `[3, 7, 10]` та `[2, 5, 8]`, яким буде об’єднаний масив після трьох операцій порівняння та копіювання?

1) `[2, 3, 5]`

2) `[2, 3, 7]`

3) `[3, 7, 2]`

4) `[2, 5, 3]`

5. Якщо реалізувати Merge Sort, де замість рекурсії використовується явний стек, чи зміняться асимптотичні складності?

1) Часова складність стане $O(n^2)$.

2) Просторова складність стане O(1).

3) Ні, асимптотичні складності залишаться O(n log n) та O(n) відповідно.

4) Часова складність покращиться до O(n).

6. Яка головна перевага Merge Sort над QuickSort?

1) Менше використання пам'яті.

2) Краща кеш-ефективність.

3) Гарантована продуктивність O(n log n) без ризику $O(n^2)$.

4) Простота реалізації.

7. Скільки операцій порівняння виконає процедура `merge` в найгіршому випадку для злиття двох масивів розміром `m` та `n`?

1) `m + n`

2) `min(m, n)`

3) `m * n`

4) `m + n - 1`

8. Щоб зробити Merge Sort стійким, під час злиття елементів `L[i]` та `R[j]`, якщо `L[i] == R[j]`, який елемент слід обрати першим?

1) Елемент з правого масиву `R[j]`.

2) Елемент з лівого масиву `L[i]`.

3) Будь-який, це не має значення.

4) Той, що має менший індекс в оригінальному масиві.

9. Що таке "природне сортування злиттям" (Natural Merge Sort)?

1) Це інша назва ітеративного сортування злиттям.

2) Це версія, яка спочатку знаходить в масиві вже існуючі відсортовані послідовності ("runs") і зливає їх.

3) Це версія, оптимізована для сортування чисел з плаваючою комою.

4) Це версія, що не використовує додаткової пам’яті.

10. Порівняно з HeapSort, Merge Sort:

1) Використовує менше пам’яті.

2) Є стійким, тоді як стандартний HeapSort — ні.

3) Має кращу часову складність у найгіршому випадку.

4) Працює швидше на малих масивах.

Відповіді:

1. 3) Глибина рекурсії дорівнює $\lceil \log_2(1000) \rceil$. Оскільки $2^9 = 512$ і $2^{10} = 1024$, глибина буде 10.

2. 2) Воно уникає накладних витрат, пов’язаних з глибокою рекурсією. Це може дати перевагу в продуктивності та уникнути помилки переповнення стеку.

3. 2) Бо його процедура злиття може послідовно обробляти дані з файлів, не завантажуючи їх повністю в пам’ять. Це дозволяє працювати з даними, що значно перевищують обсяг RAM.

4. 1). 1) `(3,2)->2` копіюється. 2) `(3,5)->3` копіюється. 3) `(7,5)->5` копіюється.

5. 3) Ні, асимптотичні складності залишаться O(n log n) та O(n) відповідно. Явний стек просто імітує системний стек рекурсії, не змінюючи фундаментальної логіки.

6. 3) Гарантована продуктивність O(n log n) без ризику $O(n^2)$. Це робить Merge Sort надійнішим для критичних застосувань.

7. 4) У найгіршому випадку останній елемент буде визначено без порівняння, коли один з масивів вже вичерпано.

8. 2) Елемент з лівого масиву `L[i]`. Це гарантує, що елементи, які були лівіше в початковому масиві, залишаться лівіше і у відсортованому.

9. 2) Це версія, яка спочатку знаходить в масиві вже існуючі відсортовані послідовності (“runs”) і зливає їх. Це робить її адаптивною.

10. 2) Є стійким, тоді як стандартний HeapSort — ні. Це ключова відмінність між ними.

Завдання для самостійної роботи

1. Реалізуйте процедуру `merge` так, щоб вона не створювала два окремі тимчасові масиви, а використовувала один допоміжний масив розміром `n`.

2. Напишіть програму, яка візуалізує процес рекурсивного ділення та подальшого злиття в Merge Sort.

3. Реалізуйте “природне сортування злиттям” (Natural Merge Sort) і порівняйте його продуктивність на майже відсортованих даних з класичним Merge Sort.

4. Дослідіть та спробуйте реалізувати K-way merge (злиття `k` відсортованих масивів одночасно), використовуючи мін-піраміду.

5. Порівняйте на практиці продуктивність Merge Sort та QuickSort на різних типах даних (випадкові, відсортовані, з дублікатами).

6. Адаптуйте Merge Sort для сортування двовимірного масиву за значеннями певного стовпця.

7. Проаналізуйте кеш-ефективність Merge Sort. Чому послідовний доступ до пам’яті в процедурі `merge` є перевагою?

8. Реалізуйте Merge Sort для сортування файлу (зовнішнє сортування), де масив не вміщується в оперативну пам’ять.

9. Дослідіть концепцію “in-place” сортування злиттям. Поясніть, чому воно є теоретично можливим, але складним і непрактичним.

10. Модифікуйте Merge Sort для сортування у зворотному порядку.

Теми лабораторних робіт

1. Реалізація та порівняльний аналіз `Top-Down` та `Bottom-Up` Merge Sort.

2. Розробка високопродуктивного сортування для зв’язних списків на основі `Merge Sort`.

3. Створення системи зовнішнього сортування для великих файлів з використанням `Merge Sort`.

4. Порівняльний аналіз стабільності: `Merge Sort` vs. `QuickSort` vs. `HeapSort`.

5. Розробка паралельної версії `Merge Sort` з використанням багатопотоковості.

6. Аналіз та реалізація `Natural Merge Sort` як адаптивного алгоритму.

7. Створення гібридного сортування `Merge Sort + Insertion Sort` для покращення продуктивності на малих підмасивах.

8. Візуалізатор дерева рекурсивних викликів та процесу злиття в `Merge Sort`.

9. Дослідження впливу розміру кешу процесора на продуктивність `Merge Sort`.

10. Реалізація `k-way merge` для злиття кількох відсортованих потоків даних у реальному часі.

9.2.2 Швидке сортування

Теоретичні відомості

Швидке сортування (Quicksort) — це надзвичайно ефективний алгоритм сортування, що базується на парадигмі “Розділяй і володарюй”. Розроблений Тоні Хоаром у 1959 році, він і досі є одним із найшвидших і найпопулярніших алгоритмів для сортування загального призначення завдяки високій середній продуктивності та можливості сортування “на місці” (in-place).

Загальний огляд: ідея та варіанти

На відміну від сортування злиттям, де основна робота відбувається на етапі “Комбінуй” (злиття), у Quicksort вся магія зосереджена на етапі “Розділяй”. Цей етап реалізується за допомогою процедури, що називається `partition` (розділення).

Велика картина Quicksort виглядає так:

1. Вибір опорного елемента (pivot): З масиву обирається один елемент. Його вибір може бути різним: перший, останній, середній або випадковий елемент.

2. Розділення (Partition): Увесь масив перевпорядковується так, що всі елементи, менші за опорний, розміщуються перед ним, а всі елементи, більші або рівні йому, — після нього. Після цього кроку опорний елемент знаходиться на своїй фінальній відсортованій позиції (це вірно для схеми Ломуто, але не завжди для Хоара).

3. Рекурсія (Conquer): Алгоритм рекурсивно застосовується до двох підмасивів, що утворилися зліва та справа від опорного елемента.

Цей процес триває, доки розмір підмасиву не стане рівним 1 або 0, що є базовим випадком рекурсії.

Ключовим елементом, від якого залежить ефективність усього алгоритму, є функція `partition`. Існує кілька популярних способів її реалізації:

- Схема розділення Хоара: Оригінальна схема, запропонована винахідником алгоритму. Вона використовує два вказівники, що рухаються назустріч один одному, і є дуже ефективною за кількістю обмінів.

- Схема розділення Ломуто: Більш проста для розуміння та реалізації схема, яка часто використовується в освітніх цілях. Вона працює за один прохід по масиву.

- Dual-Pivot Quicksort (з двома опорними елементами): Сучасна оптимізація, яка використовує два опорні елементи для розділення масиву одразу на три частини. Цей підхід зменшує кількість порівнянь і використовується, наприклад, у стандартній бібліотеці Java.

1. Схема розділення Ломуто

Це, мабуть, найпростіший для розуміння варіант. Ідея полягає в тому, щоб за один прохід розмістити всі менші елементи на початку масиву.

Алгоритм:

- Опорним елементом (`pivot`) зазвичай обирається останній елемент масиву.

- Ініціалізується вказівник `i` (межа) перед початком масиву (`left - 1`).

- Інший вказівник `j` проходить по масиву від початку до передостаннього елемента.

- Якщо `A[j]` менший або рівний за `pivot`, ми збільшуємо `i` і міняємо місцями `A[i]` та `A[j]`. Таким чином, усі елементи до індексу `i` включно є меншими за `pivot`.

- Після завершення циклу `pivot` (останній елемент) міняється місцями з `A[i+1]`, стаючи на свою фінальну позицію.

Псевдокод:

```
function LomutoPartition(array A, int left, int right)
  pivot = A[right] // Опорний елемент - останній
  i = left - 1

  for j from left to right - 1
    if A[j] <= pivot
```

```
        i = i + 1
        swap(A[i], A[j])
    end for

    swap(A[i+1], A[right])
    return i + 1 // Повертаємо індекс опорного елемента
```

Приклад роботи: `A = [4, 7, 2, 1, 5, 3]`, `pivot = 3`

- Початково: `i = -1`, `j=0`. `[4, 7, 2, 1, 5, 3]`

- `j=0`: `A[0](4) > 3`. Нічого.

- `j=1`: `A[1](7) > 3`. Нічого.

- `j=2`: `A[2](2) <= 3`. `i=0`. `swap(A[0], A[2])`. Масив: `[2, 7, 4, 1, 5, 3]`

- `j=3`: `A[3](1) <= 3`. `i=1`. `swap(A[1], A[3])`. Масив: `[2, 1, 4, 7, 5, 3]`

- `j=4`: `A[4](5) > 3`. Нічого.

- Кінець циклу. `swap(A[i+1], A[right])` -> `swap(A[2], A[5])`.

- **Результат:** `[2, 1, **3**, 7, 5, 4]`. Опорний елемент `3` на своєму місці.

Характеристика	Значення
Середній час	$O(n\log n)$
Найгірший час	$O(n^2)$
Пам’ять	$O(\log n)$
Стабільність	Ні
Переваги	Простота реалізації
Недоліки	Погана продуктивність при дублікатах

2. Схема розділення Хоара

Це оригінальний метод, запропонований автором Quicksort. Він більш ефективний, хоча трохи складніший для коректної реалізації.

Алгоритм:

- Опорний елемент (`pivot`) зазвичай обирається першим або середнім.

- Два вказівники, `i` та `j`, встановлюються на початку та в кінці масиву відповідно.

- Вказівник **i** рухається праворуч, доки не знайде елемент, більший або рівний за **pivot**.

- Вказівник **j** рухається ліворуч, доки не знайде елемент, менший або рівний за **pivot**.

- Якщо **i < j**, елементи **A[i]** та **A[j]** міняються місцями.

- Процес повторюється, доки вказівники **i** та **j** не перетнуться.

- Функція повертає індекс **j**, який є точкою розділення. На відміну від Ломуто, **pivot** після цього не обов'язково стоїть на своєму фінальному місці.

Псевдокод:

```
function HoarePartition(array A, int left, int right)
  pivot = A[left + (right - left) / 2] // Опорний - середній
  i = left - 1
  j = right + 1

  while true
    // Рухаємо лівий вказівник
    repeat
      i = i + 1
    until A[i] >= pivot

    // Рухаємо правий вказівник
    repeat
      j = j - 1
    until A[j] <= pivot

    // Якщо вказівники перетнулися, повертаємо точку розділення
    if i >= j
      return j
    end if

    // Міняємо елементи місцями
    swap(A[i], A[j])
  end while
```

Рекурсивні виклики для схеми Хоара мають бути **Quicksort(A, left, j)** та **Quicksort(A, j+1, right)**.

Приклад роботи: **A = [4, 7, 2, 1, 5, 3]**, **pivot = 2** (обраний **A[2]**)

- Початково: **i=-1, j=6**. **[4, 7, 2, 1, 5, 3]**

- **i** рухається до **A[0]**(4). **j** рухається до **A[3]**(1).

- `i<j`. `swap(A[0], A[3])`. Масив: `[1, 7, 2, 4, 5, 3]`

- `i` рухається до `A[1](7)`. `j` рухається до `A[2](2)`.

- `i<j`. `swap(A[1], A[2])`. Масив: `[1, 2, 7, 4, 5, 3]`

- `i` рухається до `A[2](7)`. `j` рухається до `A[1](2)`.

- `i >= j`. Вказівники перетнулися. Повертаємо `j=1`.

- **Результат:** `[1, 2, 7, 4, 5, 3]`. Підмасиви: `[1, 2]` та `[7, 4, 5, 3]`.

Характеристика	Значення
Середній час	$O(n\log n)$
Найгірший час	$O(n^2)$
Пам'ять	$O(\log n)$
Стабільність	Ні
Переваги	Ефективніший за Ломуто, менше обмінів
Недоліки	Складніша реалізація, pivot не на фінальній позиції

3. Quicksort з двома опорними елементами

Ця схема є суттєвою оптимізацією, яка зменшує кількість порівнянь.

Алгоритм:

- Обираються два опорні елементи, `p1` та `p2` (наприклад, перший та останній).

- Масив розділяється на три частини:

1. Елементи, менші за `p1`.

2. Елементи між `p1` та `p2` (включно).

3. Елементи, більші за `p2`.

- Це досягається за допомогою трьох вказівників: `lt` (менші за `p1`), `gt` (більші за `p2`), та `i` (поточний елемент).

- Алгоритм проходить по масиву, розподіляючи елементи по трьох секціях.

- Після розділення виконуються три рекурсивні виклики для кожної з трьох частин.

Концептуальний псевдокод:

```
function DualPivotQuicksort(array A, int left, int right)
  if left < right
    // Обираємо два pivot'и та впорядковуємо їх
```

```
    if A[left] > A[right]
        swap(A[left], A[right])
    end if
    p1 = A[left]
    p2 = A[right]

    // Розділяємо масив на три частини (логіка складна, тут пропущена
)
    // ... результатом є індекси lt та gt ...

    // Рекурсивні виклики для трьох частин
    DualPivotQuicksort(A, left, lt - 1)
    DualPivotQuicksort(A, lt + 1, gt - 1)
    DualPivotQuicksort(A, gt + 1, right)
  end if
end function
```

Приклад роботи: `A = [4, 7, 2, 1, 5, 3]`, `p1=3, p2=4`

- Початково: `[4, 7, 2, 1, 5, 3]`

- **Результат розділення:** `[2, 1, 3, 4, 7, 5]`.

- Частина 1: `[2, 1]` (менші за **3**)

- Частина 2: `[]` (між **3** та **4**)

- Частина 3: `[7, 5]` (більші за **4**)

- Далі рекурсія для кожної з частин.

Характеристика	Значення
Середній час	$O(n\log n)$ (швидший за класичний)
Найгірший час	$O(n^2)$
Пам’ять	$O(\log n)$
Стабільність	Ні
Переваги	Найвища продуктивність на практиці, стандарт в Java
Недоліки	Складність реалізації

Практичні аспекти та порівняння

Вибір схеми розділення та деякі оптимізації можуть кардинально вплинути на продуктивність Quicksort.

- Коли що обирати:

- Схема Ломуто: Ідеальна для освітніх цілей або коли потрібна найпростіша реалізація.

- Схема Хоара: Кращий вибір для більшості практичних реалізацій, якщо ви не пишете системну бібліотеку. Вона швидша і краще обробляє дублікати.

- Dual-Pivot: Найкращий вибір для високопродуктивних бібліотек загального призначення, де навіть невеликий відсоток прискорення має значення.

- Вибір опорного елемента (Pivot):

  - Щоб уникнути найгіршого випадку $O(n^2)$ (який виникає на відсортованих або майже відсортованих даних при виборі першого/останнього елемента), використовують кращі стратегії:

    - Випадковий елемент: Простий та ефективний спосіб уникнути систематичних “поганих” вхідних даних.

    - Медіана трьох (Median-of-Three): Опорним елементом обирається медіана першого, середнього та останнього елементів масиву. Це надійно захищає від найгірших випадків на впорядкованих даних.

- Гібридизація з сортуванням вставкою:

  - Для дуже малих підмасивів (зазвичай, розміром менше 10-20 елементів) накладні витрати на рекурсію Quicksort стають невиправданими.

  - Ефективна оптимізація полягає в тому, щоб припинити рекурсію, коли розмір підмасиву стає меншим за певний поріг.

  - Після завершення Quicksort весь масив проходиться один раз сортуванням вставкою, яке дуже швидко ($O(nk)$) впорядковує такі “майже відсортовані” масиви.

Область застосування

Quicksort є де-факто стандартом для сортування в пам’яті загального призначення.

- Стандартні бібліотеки мов програмування: Багато реалізацій **`sort()`** використовують Quicksort або його гібридні версії (наприклад, Introsort, який є гібридом Quicksort, Heapsort та Insertion Sort).

- Прикладні програми: Будь-де, де потрібне швидке сортування масиву і стабільність не є вимогою.

- Системне програмування: Часто використовується завдяки своїй `in-place` природі.

Переваги та недоліки

Переваги:

- Висока середня швидкість: На практиці часто є найшвидшим алгоритмом сортування.

- Сортування “на місці” (in-place): Потребує лише $O(\log n)$ додаткової пам’яті для стеку рекурсії.

- Хороша кеш-ефективність: Послідовний доступ до елементів масиву під час розділення добре працює з кешем процесора.

Недоліки:

- Найгірший випадок $O(n^2)$: Хоча рідкісний на практиці при правильній реалізації, він є теоретичним ризиком.

- Нестійкість: Не зберігає відносний порядок однакових елементів.

- Складність реалізації: Особливо для схеми Хоара та Dual-Pivot, де легко припуститися помилки з індексами.

Контрольні запитання

1. У чому полягає основна відмінність між підходом “Розділяй і володарюй” в Quicksort та Merge Sort?

2. Поясніть різницю між схемами розділення Хоара та Ломуто. Яка з них, як правило, ефективніша і чому?

3. Чому вибір опорного елемента є критично важливим для Quicksort? Опишіть стратегію “медіана трьох”.

4. Навіщо використовувати гібридний підхід з перемиканням на сортування вставкою для малих підмасивів?

5. Поясніть, чому Quicksort є нестабільним алгоритмом.

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та порівняльний аналіз схем розділення в алгоритмі Quicksort.

2. Мета:

- Реалізувати алгоритм Quicksort з використанням схем розділення Ломуто та Хоара.

- Дослідити вплив вибору опорного елемента на продуктивність.

- Реалізувати гібридний підхід із сортуванням вставкою.

3. Завдання:

- Реалізувати Quicksort зі схемою Ломуто.

- Реалізувати Quicksort зі схемою Хоара.

- Додати до реалізацій вибір опорного елемента за стратегією “медіана трьох”.

- Провести експерименти на різних типах даних: випадкових, відсортованих, відсортованих у зворотному порядку.

- Реалізувати гібридну версію (з переходом на Insertion Sort для `n < 16`) і порівняти її продуктивність.

4. Опис: Лабораторна робота фокусується на практичних аспектах реалізації Quicksort. Студент повинен не просто написати код, а й на практиці побачити, як різні схеми розділення та стратегії вибору опори впливають на швидкість, особливо у найгірших випадках (наприклад, на відсортованих даних).

5. Порядок виконання:

- Реалізувати дві версії `partition` (Ломуто та Хоара) та відповідні функції `Quicksort`.

- Протестувати базові версії (з вибором останнього/першого елемента як опори) на відсортованому масиві, щоб побачити складність $O(n^2)$.

- Додати стратегію “медіана трьох” і перевірити, як вона вирішує проблему на відсортованих даних.

- Порівняти час виконання схем Ломуто та Хоара на великих випадкових масивах.

- Додати гібридну оптимізацію та виміряти приріст продуктивності.

- Результати звести у таблиці та зробити висновки.

6. Результат:

- Програма з кількома реалізаціями Quicksort.

- Звіт, що містить аналіз продуктивності, порівняння схем та ефективності оптимізацій.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
    - Функції розділення `LomutoPartition` та `HoarePartition`.
    - Логіка вибору опорного елемента (“медіана трьох”).
    - Гібридний підхід.
- Продемонструвати роботу додатку:
    - Виконання на відсортованому масиві для демонстрації проблеми $O(n^2)$ та її вирішення.
    - Порівняння швидкості різних версій на великому масиві.
- Бути готовим відповісти на питання:
    - Яка складність Quicksort і від чого вона залежить?
    - У чому різниця в рекурсивних викликах після розділення Хоара та Ломуто?
    - Чому Dual-Pivot Quicksort швидший?
    - Чому Quicksort не використовують для зовнішнього сортування?

Тести

1. Яка середня часова складність Quicksort?

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) O(log n)

2. Найгірший випадок для Quicksort ($O(n^2)$) виникає, коли:

1) Масив складається з однакових елементів.

2) Опорний елемент на кожному кроці виявляється найменшим або найбільшим.

3) Масив повністю випадковий.

4) Використовується схема розділення Хоара.

3. Яка схема розділення вважається простішою для реалізації, але менш ефективною?

1) Схема Хоара.

2) Схема Ломуто.

3) Dual-Pivot.

4) Схема Седжвіка.

4. Чому Quicksort зазвичай не є стійким сортуванням?

1) Через рекурсивні виклики.

2) Через обміни елементів, що знаходяться на великій відстані один від одного.

3) Через використання додаткової пам’яті.

4) Він є стійким.

5. Стратегія “медіана трьох” використовується для:

1) Прискорення процедури `partition`.

2) Зменшення використання пам’яті.

3) Зменшення ймовірності виникнення найгіршого випадку $O(n^2)$.

4) Забезпечення стабільності алгоритму.

Відповіді:

1. b)
2. b)
3. b)
4. b)
5. c)

Тести підвищеної складності

1. Після виконання `LomutoPartition` для масиву `[8, 1, 6, 2, 5]` з `pivot=5`, яким буде масив?

1) `[1, 2, 5, 6, 8]`

2) `[1, 2, 8, 6, 5]`

3) `[2, 1, 5, 8, 6]`

4) `[1, 2, 5, 8, 6]`

2. Після виконання `HoarePartition` для `[8, 1, 6, 2, 5]` з `pivot=5` (середній елемент 6, але для простоти візьмемо 5), яким може бути результат?

1) `[1, 2, 5, 6, 8]`

2) `[2, 1, 5, 8, 6]`

3) Обидва варіанти можливі, оскільки позиція `pivot` не фіксована.

4) `[5, 1, 6, 2, 8]`

3. Чому Dual-Pivot Quicksort теоретично швидший за класичний?

1) Він використовує менше пам'яті.

2) Він розділяє масив на три частини, що в середньому зменшує загальну кількість порівнянь.

3) Він завжди уникає найгіршого випадку.

4) Він є стабільним.

4. Що таке Introsort (інтроспективне сортування)?

1) Це інша назва для Quicksort.

2) Гібридний алгоритм, що починає з Quicksort, але перемикається на Heapsort, якщо глибина рекурсії стає завеликою, щоб уникнути $O(n^2)$.

3) Алгоритм, що використовує лише сортування вставкою.

4) Алгоритм, оптимізований для сортування рядків.

5. Яка просторова складність Quicksort у найгіршому випадку (без оптимізації хвостової рекурсії)?

1) O(1)

2) O(log n)

3) O(n)

4) $O(n^2)$

6. У схемі Хоара, якщо після циклу `i` та `j` перетнулися (`i >= j`), функція повертає `j`. Які будуть межі для рекурсивних викликів?

1) `(left, j-1)` та `(j, right)`

2) `(left, j)` та `(j+1, right)`

3) `(left, i-1)` та `(i, right)`

4) `(left, j-1)` та `(j+1, right)`

7. Якщо масив складається з однакових елементів, як поведе себе Quicksort зі схемою Ломуто?

1) Дуже швидко, складність O(n).

2) Дуже повільно, складність $O(n^2)$.

3) Час роботи O(n log n).

4) Не відсортує масив.

8. Який алгоритм є кращим для сортування зв'язного списку?

1) Quicksort, бо він найшвидший.

2) Heapsort, бо він in-place.

3) Merge Sort, бо він не вимагає довільного доступу і може бути реалізований з O(1) додаткової пам'яті.

4) Insertion Sort, бо він простий.

9. Оптимізація хвостової рекурсії в Quicksort допомагає:

1) Зменшити час роботи в середньому випадку.

2) Зменшити використання пам'яті стеком з O(n) до O(log n) у найгіршому випадку.

3) Зробити алгоритм стійким.

4) Прискорити процедуру `partition`.

10. Яке твердження про Quicksort є хибним?

1) Це сортування методом порівняння.

2) Це сортування "на місці" (з урахуванням стеку рекурсії).

3) Його продуктивність сильно залежить від вибору опорного елемента.

4) Він гарантує часову складність O(n log n) для будь-яких вхідних даних.

Відповіді:

1. 4) Після циклу `j`: `[1, 2, 8, 6, 5]`. Фінальний обмін `swap(A[i+1], A[right])` -> `swap(A[2], A[4])`. Результат: `[1, 2, 5, 8, 6]`.

2. 2) Схема Хоара не гарантує фінальну позицію pivot. Вона лише гарантує, що елементи зліва від `j` менші або рівні, а справа - більші або рівні.

3. 2) Він розділяє масив на три частини, що в середньому зменшує загальну кількість порівнянь. Це більш ефективно використовує інформацію, отриману від порівнянь.

4. 2) Гібридний алгоритм, що починає з Quicksort, але перемикається на Heapsort, якщо глибина рекурсії стає завеликою, щоб уникнути $O(n^2)$. Також зазвичай використовує Insertion Sort для малих підмасивів.

5. 3) У найгіршому випадку (на відсортованому масиві) розділення буде незбалансованим (на `n-1` та `0` елементів), що призведе до глибини рекурсії `n`.

6. 2) Це правильний спосіб уникнути нескінченної рекурсії для схеми Хоара.

7. 2) Схема Ломуто (`A[j] <= pivot`) призведе до незбалансованого розділення на `n-1` і `0` елементів на кожному кроці.

8. 3) Merge Sort, бо він не вимагає довільного доступу і може бути реалізований з O(1) додаткової пам'яті. Quicksort потребує швидкого доступу до елементів, що для зв'язного списку є операцією O(n).

9. 2) Зменшити використання пам'яті стеком з O(n) до O(log n) у найгіршому випадку. Це досягається шляхом ітеративної обробки більшого підмасиву.

10. 4) Він гарантує часову складність O(n log n) для будь-яких вхідних даних. Це хибне твердження; найгірший випадок - $O(n^2)$. Гарантію $O(n\log n)$ дає Merge Sort.

Завдання для самостійної роботи

1. Реалізуйте Quicksort зі схемою Ломуто та стратегією вибору випадкового опорного елемента.

2. Реалізуйте Quicksort зі схемою Хоара та стратегією "медіана трьох".

3. Напишіть програму, яка візуалізує процес розділення для схем Ломуто та Хоара.

4. Порівняйте на практиці кількість обмінів для схем Ломуто та Хоара на великих масивах.

5. Реалізуйте оптимізацію хвостової рекурсії для Quicksort.

6. Створіть "антитест" для Quicksort з простою стратегією вибору опори (наприклад, перший елемент), який призводить до складності $O(n^2)$.

7. Дослідіть та реалізуйте 3-way Quicksort, який ефективно обробляє масиви з великою кількістю дублікатів, розділяючи масив на три частини: менші, рівні та більші за опору.

8. Порівняйте продуктивність вашої реалізації Quicksort з вбудованою функцією сортування у вашій мові програмування.

9. Напишіть тести для перевірки коректності меж рекурсивних викликів для схем Хоара та Ломуто.

10. Дослідіть, як Quicksort поводиться на масивах, де всі елементи однакові, для різних схем розділення.

Теми лабораторних робіт

1. Порівняльний аналіз продуктивності схем розділення Ломуто та Хоара.

2. Вплив стратегій вибору опорного елемента (перший, випадковий, медіана трьох) на продуктивність Quicksort.

3. Реалізація та аналіз ефективності гібридного **`Quicksort + Insertion Sort`**.

4. Розробка та тестування **`3-way Quicksort`** для масивів з великою кількістю дублікатів.

5. Аналіз та реалізація **`Dual-Pivot Quicksort`**.

6. Створення візуалізатора, що демонструє процес роботи різних варіантів Quicksort.

7. Дослідження кеш-ефективності Quicksort та порівняння з Merge Sort.

8. Реалізація **`Introsort`** (гібрид Quicksort, Heapsort, Insertion Sort).

9. Аналіз просторової складності Quicksort: вимірювання глибини рекурсії та реалізація з оптимізацією хвостової рекурсії.

10. Застосування Quicksort для вирішення задачі пошуку k-ої статистики (алгоритм Quickselect).

9.2.3 Пірамідальне сортування

Теоретичні відомості

Пірамідальне сортування (Heapsort) — це витончений та ефективний алгоритм сортування, що базується на порівняннях. Його унікальність полягає в тому, що він поєднує переваги двох інших ефективних алгоритмів: гарантовану часову складність O(n log n), як у сортування злиттям, та виконання “на місці” (in-place) з просторовою складністю O(1), як у Quicksort. Для своєї роботи Heapsort використовує спеціалізовану структуру даних — бінарну піраміду (binary heap).

Структура даних: Бінарна піраміда (Binary Heap)

Перш ніж зрозуміти сортування, необхідно зрозуміти структуру даних, на якій воно базується.

Бінарна піраміда — це бінарне дерево, що задовольняє двом умовам:

1. Умова повноти: Усі рівні дерева, крім, можливо, останнього, повністю заповнені, а останній рівень заповнюється зліва направо. Це дозволяє компактно зберігати піраміду в звичайному масиві.

2. Властивість піраміди: Значення в кожному вузлі задовольняє певній умові відносно значень у його дочірніх вузлах. Існує два типи пірамід:

- Max-Heap (Макс-піраміда): Значення кожного вузла більше або дорівнює значенням його дітей. Отже, найбільший елемент завжди знаходиться в корені дерева.

- Min-Heap (Мін-піраміда): Значення кожного вузла менше або дорівнює значенням його дітей. Найменший елемент — у корені.

Для сортування за зростанням ми будемо використовувати Max-Heap.

Представлення піраміди у вигляді масиву: Завдяки умові повноти, піраміду можна ефективно представити у вигляді масиву `A` без явного зберігання вказівників.

- Корінь дерева знаходиться в `A[0]`.

- Для вузла з індексом `i`:

- Його батько знаходиться за індексом: `parent(i) = floor((i - 1) / 2)`

- Його ліва дитина знаходиться за індексом: `left(i) = 2*i + 1`

- Його права дитина знаходиться за індексом: `right(i) = 2*i + 2`

Алгоритм Heapsort: Дві фази

Процес пірамідального сортування складається з двох основних фаз:

Фаза 1: Побудова Макс-піраміди (Build-Max-Heap) Перший крок — перетворити вхідний невідсортований масив на макс-піраміду. Це можна зробити ефективно за час $O(n)$ за допомогою процедури, яка починається з останнього вузла, що має дочірні елементи (`floor(n/2) - 1`), і рухається до кореня, викликаючи для кожного вузла процедуру `heapify`.

Фаза 2: Сортування шляхом вилучення елементів Після побудови макс-піраміди найбільший елемент масиву знаходиться в корені (`A[0]`).

1. Поміняти місцями кореневий елемент `A[0]` з останнім елементом піраміди `A[n-1]`. Тепер найбільший елемент знаходиться в кінці масиву, на своїй фінальній позиції.

2. “Видалити” цей елемент з піраміди, зменшивши її розмір на 1.

3. Після обміну новий корінь **A[0]** може порушувати властивість макс-піраміди. Щоб її відновити, викликаємо процедуру **heapify** для кореня (індекс 0) на зменшеній піраміді.

4. Кроки 1-3 повторюються **n-1** разів. Після кожного кроку наступний найбільший елемент переміщується на своє місце.

Ключові процедури: **heapify** та **buildMaxHeap**

heapify (також відома як **siftDown**) Це основна процедура для підтримки властивості піраміди. Вона приймає на вхід індекс вузла **i** та розмір піраміди. Припускається, що бінарні дерева, коренями яких є ліва та права дитини вузла **i**, вже є макс-пірамідами.

Псевдокод heapify:

```
// A - масив, n - розмір піраміди, i - індекс кореня піддерева
function heapify(array A, int n, int i)
  largest = i      // Ініціалізуємо найбільший елемент як корінь
  left = 2*i + 1   // індекс лівої дитини
  right = 2*i + 2  // індекс правої дитини

  // Якщо ліва дитина існує і більша за корінь
  if left < n and A[left] > A[largest]
    largest = left
  end if

  // Якщо права дитина існує і більша за поточний найбільший
  if right < n and A[right] > A[largest]
    largest = right
  end if

  // Якщо найбільший елемент - не корінь
  if largest != i
    swap(A[i], A[largest])
    // Рекурсивно викликаємо heapify для піддерева, яке було змінено
    heapify(A, n, largest)
  end if
end function
```

buildMaxHeap Ця процедура будує макс-піраміду з невпорядкованого масиву.

Псевдокод buildMaxHeap:

```
function buildMaxHeap(array A)
  n = length(A)
  // Починаємо з останнього вузла, що не є листком, і рухаємося до ко
```

```
реня
  for i from floor(n/2) - 1 down to 0
    heapify(A, n, i)
  end for
end function
```

Повний псевдокод Heapsort

```
function HeapSort(array A)
  n = length(A)

  // Фаза 1: Побудова макс-піраміди
  buildMaxHeap(A)

  // Фаза 2: Вилучення елементів з піраміди
  for i from n - 1 down to 1
    // Переміщуємо поточний корінь (найбільший елемент) в кінець
    swap(A[0], A[i])

    // Викликаємо heapify для зменшеної піраміди (розмір піраміди теп
ер i)
    heapify(A, i, 0)
  end for
end function
```

Приклад роботи

Розглянемо сортування масиву `A = [4, 10, 3, 5, 1]`. `n=5`.

Фаза 1: `buildMaxHeap`

- Починаємо з `i = floor(5/2) - 1 = 1`.

- `heapify(A, 5, 1)`: `A[1]=10`. Діти: `A[3]=5`, `A[4]=1`. `10` є найбільшим. Нічого не змінюється.

- `heapify(A, 5, 0)`: `A[0]=4`. Діти: `A[1]=10`, `A[2]=3`. `largest` стає `1`.

- `swap(A[0], A[1])`. Масив: `[10, 4, 3, 5, 1]`.

- Рекурсивний виклик `heapify(A, 5, 1)`. `A[1]=4`. Діти `A[3]=5`, `A[4]=1`. `largest` стає `3`.

- `swap(A[1], A[3])`. Масив: `[10, 5, 3, 4, 1]`.

- Піраміду побудовано: `[10, 5, 3, 4, 1]`.

Фаза 2: Сортування

1. `i = 4`: `swap(A[0], A[4])` -> `[1, 5, 3, 4, **10**]`.

- `heapify(A, 4, 0)`. Результат: `[5, 4, 3, 1, **10**]`.

2. `i = 3`: `swap(A[0], A[3])` -> `[1, 4, 3, **5**, 10]`.

- `heapify(A, 3, 0)`. Результат: `[4, 1, 3, **5**, 10]`.

3. `i = 2`: `swap(A[0], A[2])` -> `[3, 1, **4**, 5, 10]`.

- `heapify(A, 2, 0)`. Результат: `[3, 1, **4**, 5, 10]`.

4. `i = 1`: `swap(A[0], A[1])` -> `[1, **3**, 4, 5, 10]`.

- `heapify(A, 1, 0)`. Результат: `[1, **3**, 4, 5, 10]`.

Фінальний відсортований масив: `[1, 3, 4, 5, 10]`.

Аналіз складності

- Часова складність: O(n log n).

- Це справедливо для найкращого, середнього та найгіршого випадків.

- Побудова піраміди (`buildMaxHeap`): $O(n)$.

- Фаза сортування: `n-1` разів викликається `heapify`, складність якої $O(\log k)$ на піраміді розміром `k`. Сумарна складність цієї фази — $O(n \log n)$.

- Загальна: $O(n) + O(n \log n) = O(n \log n)$.

- Просторова складність: O(1).

- Алгоритм працює “на місці”, не вимагаючи додаткового простору, пропорційного розміру масиву.

Властивості алгоритму

- Нестійкість**:** Heapsort є нестійким. Обміни між коренем та останнім елементом можуть порушити відносний порядок однакових елементів.

- Не адаптивний: Виконує однакову кількість операцій незалежно від того, наскільки впорядкованим є початковий масив.

- Погана кеш-ефективність: Під час `heapify` алгоритм “стрибає” по масиву між батьківськими та дочірніми елементами, які можуть знаходитись далеко один від одного в пам’яті. Це призводить до великої кількості кеш-промахів.

Область застосування

Heapsort є чудовим вибором, коли потрібні гарантії продуктивності та ефективне використання пам’яті.

- Критичні системи: Коли найгірший випадок $O(n^2)$ у Quicksort є неприйнятним ризиком, а додаткова пам’ять $O(n)$ для Merge Sort недоступна.

- Вбудовані системи та ядра операційних систем: Linux Kernel використовує Heapsort. Його `in-place` природа та гарантована складність роблять його ідеальним для таких середовищ.

- Реалізація черг з пріоритетом: Структура даних “піраміда” лежить в основі ефективної реалізації черг з пріоритетом.

- Компонент інших алгоритмів: Використовується в Introsort для запобігання найгіршому випадку Quicksort.

Переваги та недоліки

Переваги:

- Гарантована часова складність O(n log n).

- Просторова складність O(1): працює “на місці”.

- Універсальність: добре працює для будь-яких вхідних даних.

Недоліки:

- Нестійкість.

- Погана локальність посилань (кеш-ефективність): На практиці часто повільніший за добре реалізований Quicksort для типових даних через кеш-промахи.

- Не адаптивний.

Контрольні запитання

1. Що таке бінарна піраміда і які дві властивості її визначають?

2. Поясніть, як піраміда представляється у вигляді масиву. Напишіть формули для знаходження батька та дітей для вузла `i`.

3. Опишіть дві основні фази пірамідального сортування.

4. Яке призначення процедури `heapify`?

5. Чому Heapsort, на відміну від Quicksort, має гарантовану складність $O(n \log n)$, але на практиці часто повільніший?

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз пірамідального сортування.

2. Мета:

- Реалізувати ключові процедури для роботи з пірамідою: `heapify`, `buildMaxHeap`.

- Реалізувати алгоритм Heapsort.

- Порівняти продуктивність Heapsort з Quicksort.

3. Завдання:

- Реалізувати функцію `heapify`.

- На її основі реалізувати функцію `buildMaxHeap`.

- Реалізувати основну функцію `HeapSort`.

- Написати програму-бенчмарк для порівняння часу роботи Heapsort та Quicksort на великих масивах випадкових даних.

- Написати функцію, що перевіряє, чи є масив макс-пірамідою.

4. Опис: Лабораторна робота вимагає глибокого розуміння структури даних “піраміда”. Студент повинен реалізувати алгоритм з нуля, починаючи з базових операцій. Порівняння з Quicksort на практиці дозволить зрозуміти різницю між теоретичною гарантованою продуктивністю та реальною швидкістю, на яку впливає кеш-ефективність.

5. Порядок виконання:

- Реалізувати функцію `heapify(A, n, i)`. Ретельно її протестувати.

- Реалізувати `buildMaxHeap(A)`, використовуючи `heapify`. Перевірити її роботу за допомогою функції-валідатора піраміди.

- Реалізувати `HeapSort(A)`.

- Реалізувати Quicksort (можна взяти з попередньої роботи).

- Провести заміри часу для обох алгоритмів на масивах розміром 100 000, 1 000 000, 10 000 000 елементів.

- Результати звести в таблицю та проаналізувати.

6. Результат:

- Програма, що коректно реалізує Heapsort.

- Звіт з порівняльним аналізом продуктивності Heapsort та Quicksort, та висновками щодо їх практичного застосування.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
    - Процедура `heapify`.
    - Процедура `buildMaxHeap`.
    - Основна функція `HeapSort`.

- Продемонструвати роботу додатку:
    - Побудова піраміди з невеликого масиву та її вивід.
    - Покрокове сортування піраміди.
    - Порівняння часу роботи з Quicksort.
- Бути готовим відповісти на питання:
    - Яка складність кожної з фаз Heapsort?
    - Чому Heapsort не є стабільним?
    - Чому Heapsort має погану кеш-ефективність?
    - У яких випадках Heapsort є кращим вибором, ніж Merge Sort або QuickSort?

Тести

1. Яку структуру даних використовує Heapsort?

1) Червоно-чорне дерево.

2) Зв’язний список.

3) Бінарну піраміду.

4) Хеш-таблицю.

2. Яка просторова складність Heapsort?

1) O(1)

2) O(log n)

3) O(n)

4) O(n log n)

3. Яка властивість є істинною для макс-піраміди?

1) Кожен вузол менший за своїх дітей.

2) Найбільший елемент знаходиться в одному з листків.

3) Кожен вузол більший або рівний за своїх дітей.

4) Дерево має бути ідеально збалансованим.

4. Що відбувається на кожному кроці другої фази Heapsort?

1) Два сусідні елементи міняються місцями.

2) Корінь піраміди міняється місцями з останнім елементом, і піраміда відновлюється.

3) Масив ділиться на дві частини.

4) Знаходиться мінімальний елемент і ставиться на початок.

5. Основна перевага Heapsort над Quicksort полягає у:

1) Кращій середній продуктивності.

2) Стабільності.

3) Гарантованій часовій складності O(n log n) без ризику $O(n^2)$.

4) Кращій кеш-ефективності.

Відповіді:

1. c)
2. a)
3. c)
4. b)
5. c)

Тести підвищеної складності

1. Для масиву **A**, що представляє піраміду, якщо вузол знаходиться за індексом **i=3**, де знаходиться його батько?

1) індекс 0

2) індекс 1

3) індекс 2

4) індекс 7

2. Часова складність ефективної побудови піраміди (**buildMaxHeap**) з **n** елементів становить:

1) $O(n^2)$

2) O(n log n)

3) O(n)

4) O(log n)

3. Чому Heapsort не є стабільним сортуванням?

1) Через рекурсивні виклики **heapify**.

2) Через те, що процедура **buildMaxHeap** порушує порядок.

3) Через обмін кореневого елемента з останнім елементом піраміди на кожному кроці сортування.

4) Він є стабільним.

4. Після побудови макс-піраміди з масиву `[3, 5, 8, 10, 1, 2]`, який елемент буде в `A[0]`?

1) 3

2) 8

3) 10

4) 1

5. Що таке Introsort?

1) Це інша назва для Heapsort.

2) Гібридний алгоритм, що використовує Heapsort для уникнення найгіршого випадку Quicksort.

3) Версія Heapsort, оптимізована для малих масивів.

4) Алгоритм, що будує тернарну піраміду замість бінарної.

6. На масиві, що вже відсортований у зворотному порядку, Heapsort працюватиме:

1) За час O(n).

2) За час O(n log n).

3) За час $O(n^2)$.

4) Швидше, ніж на випадкових даних.

7. Якщо елементи вставляти в порожню піраміду по одному, яка буде загальна складність побудови піраміди з `n` елементів?

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) O(log n)

8. Який алгоритм НЕ використовує Heapsort як компонент для покращення надійності?

1) Introsort.

2) Стандартне сортування в деяких реалізаціях C++ STL.

3) Merge Sort.

4) Сортування в ядрі Linux.

9. Після першого кроку сортувальної фази (один обмін та один виклик `heapify`) для піраміди `[20, 15, 8, 10, 5, 7]`, яким буде елемент в `A[0]`?

1) 20

2) 8

3) 15

4) 10

10. Головна причина, чому Heapsort на практиці повільніший за Quicksort, це:

1) Більша кількість обмінів.

2) Більша кількість порівнянь.

3) Погана локальність посилань та кеш-промахи.

4) Складність обчислення індексів дітей/батьків.

Відповіді:

1. 2) `floor((3 - 1) / 2) = floor(1) = 1`.

2. 3) Хоча це не очевидно, точний математичний аналіз показує лінійну складність для bottom-up побудови.

3. 3) Через обмін кореневого елемента з останнім елементом піраміди на кожному кроці сортування. Цей далекий обмін може змінити відносний порядок рівних елементів.

4. 3) Після `buildMaxHeap` на вершині піраміди (`A[0]`) буде найбільший елемент масиву.

5. 2) Гібридний алгоритм, що використовує Heapsort для уникнення найгіршого випадку Quicksort.

6. 2) Heapsort не є адаптивним, тому час роботи не залежить від початкового стану масиву.

7. 2) Кожна з `n` вставок може вимагати до $O(\log k)$ операцій `siftUp` на піраміді розміром `k`.

8. 3) Merge Sort є самодостатнім алгоритмом і не потребує Heapsort.

9. 3) `swap(A[0], A[5])` -> `[7, 15, 8, 10, 5, **20**]`. `heapify(A, 5, 0)` підніме `15` на місце кореня.

10. 3) Погана локальність посилань та кеш-промахи. “Стрибки” по пам’яті між батьком та дітьми є дорогими на сучасних архітектурах.

Завдання для самостійної роботи

1. Реалізуйте Heapsort з використанням мін-піраміди (Min-Heap) для сортування масиву за спаданням.

2. Напишіть програму, що візуалізує масив у вигляді бінарної піраміди та показує зміни під час `heapify` та сортування.

3. Порівняйте на практиці ефективність побудови піраміди за $O(n)$ (`buildMaxHeap`) та за $O(n \log n)$ (шляхом `n` вставок).

4. Реалізуйте структуру даних “Черга з пріоритетом” на основі бінарної піраміди.

5. Дослідіть та реалізуйте тернарну піраміду (де кожен вузол має трьох дітей) та відповідний Heapsort. Порівняйте продуктивність.

6. Проаналізуйте кількість порівнянь та обмінів у Heapsort на різних типах даних.

7. Реалізуйте Heapsort для сортування зв’язного списку. Чи є це практичним?

8. Напишіть тести для перевірки коректності вашої реалізації `heapify` на різних граничних випадках.

9. Дослідіть алгоритм Smoothsort, який є варіацією Heapsort, але має адаптивну складність $O(n)$ для вже відсортованих даних.

10. Порівняйте кеш-ефективність Heapsort та Merge Sort шляхом підрахунку кеш-промахів (можна використовувати спеціалізовані інструменти типу `perf` або `valgrind`).

Теми лабораторних робіт

1. Реалізація та порівняльний аналіз продуктивності `Heapsort`, `Quicksort` та `Merge Sort`.

2. Розробка та тестування високопродуктивної черги з пріоритетом на основі бінарної піраміди.

3. Аналіз кеш-ефективності та локальності посилань в `Heapsort`.

4. Реалізація `Introsort` (гібрид `Quicksort` та `Heapsort`).

5. Візуалізатор для демонстрації побудови піраміди та процесу пірамідального сортування.

6. Порівняння бінарної та d-арної пірамід (де кожен вузол має `d` дітей) та їх застосування в `Heapsort`.

7. Застосування `Heapsort` для вирішення задачі пошуку `k` найбільших елементів у масиві.

8. Дослідження та реалізація `Smoothsort` як адаптивної альтернативи `Heapsort`.

9. Аналіз стабільності: розробка стабільної версії `Heapsort` та оцінка її накладних витрат.

10. Оцінка продуктивності `Heapsort` при сортуванні масивів об'єктів з дорогими операціями порівняння.

9.3. Сортування за лінійний час: коли порівняння не потрібні

Досі ми розглядали алгоритми, що базуються на порівняннях, для яких теоретична межа швидкості складає $O(n\log n)$. Це фундаментальне обмеження, яке неможливо подолати, якщо єдиною інформацією про дані є результат порівняння двох елементів. Але що, якщо ми знаємо про дані щось більше?

Цей розділ відкриває двері у світ непорівняльних сортувань — алгоритмів, які досягають дивовижної лінійної часової складності $O(n)$, експлуатуючи специфічні властивості вхідних даних. Замість того, щоб питати "який елемент більший?", ці методи використовують інформацію про значення, діапазон або структуру самих ключів для їхнього прямого розподілу по позиціях. Ми розглянемо чотири потужні техніки:

- **Сортування підрахунком (Counting Sort):** Ідеальний для сортування цілих чисел у невеликому діапазоні. Цей алгоритм просто підраховує кількість входжень кожного елемента, а потім використовує цю інформацію для прямого розміщення елементів на їхні фінальні позиції.

- **Кишенькове (коміркове) сортування (Bucket Sort):** Ефективний для даних, що рівномірно розподілені у певному діапазоні. Алгоритм розподіляє елементи по скінченній кількості "кишень" (buckets), сортує кожну кишеню окремо (часто за допомогою сортування вставкою), а потім збирає результати.

- **Порозрядне сортування (Radix Sort):** Геніальний метод, що сортує числа, обробляючи їх по розрядах (цифрах). Ми розглянемо два підходи: LSD (Least Significant Digit), що починає з наймолодшого розряду, та MSD (Most Significant Digit), що починає зі старшого. Це один з найшвидших алгоритмів для сортування 32-х та 64-х розрядних цілих чисел.

- Flashsort: Розподільчий алгоритм, який використовує класифікацію для оцінки фінальної позиції кожного елемента, досягаючи лінійного часу для рівномірно розподілених даних з мінімальними вимогами до пам'яті.

Вивчення цих алгоритмів покаже, що іноді найкращий спосіб вирішити задачу — це змінити правила гри. Хоча вони не є універсальними, у правильних умовах ці методи демонструють неперевершену швидкість.

9.3.1 Сортування підрахунком

Теоретичні відомості

Сортування підрахунком — це перший алгоритм у нашому огляді, що кардинально відрізняється від попередніх. Він не базується на порівняннях, що дозволяє йому подолати теоретичний бар'єр швидкості $O(n \log n)$. Натомість, він використовує інформацію про значення самих елементів для досягнення лінійної часової складності.

Ключова умова для застосування цього алгоритму: вхідні дані мають бути цілими числами (або такими, що можна однозначно відобразити в цілі числа) в обмеженому діапазоні.

Основний принцип роботи:

Замість того, щоб порівнювати елементи між собою, алгоритм визначає для кожного елемента `x` його фінальну позицію шляхом підрахунку кількості елементів, менших за `x`.

Розглянемо стабільну версію алгоритму, яка є найбільш універсальною і використовується як частина більш складних сортувань (наприклад, Radix Sort).

Алгоритм складається з кількох кроків:

1. Знаходження діапазону: Знайти мінімальне (`min`) та максимальне (`max`) значення у вхідному масиві `A`. Це необхідно для визначення розміру допоміжного масиву. Розмір діапазону `k = max - min + 1`.

2. Створення масиву лічильників: Створити допоміжний масив `count` розміром `k`, ініціалізований нулями.

3. Підрахунок частот: Пройти по вхідному масиву `A`. Для кожного елемента `A[i]` збільшити лічильник `count[A[i] - min]` на одиницю. Після цього кроку `count[j]` містить кількість входжень числа `j + min` у масиві `A`.

4. Обчислення кумулятивних сум: Модифікувати масив `count` так, щоб кожен його елемент `count[j]` містив суму всіх попередніх лічильників. Тобто, `count[j]` тепер вказує, скільки елементів у `A` є меншими або рівними `j + min`. Цей крок є ключовим для визначення фінальних позицій елементів.

5. Формування результуючого масиву: Створити вихідний масив `B` такого ж розміру, як `A`. Пройти по вхідному масиву `A` у зворотному порядку (з кінця до початку). Для кожного елемента `A[i]`:

- Його фінальна позиція у відсортованому масиві визначається значенням `count[A[i] - min] - 1`.

- Помістити `A[i]` у `B` на знайдену позицію.

- Зменшити лічильник `count[A[i] - min]` на одиницю. Зворотний прохід та зменшення лічильника гарантують стабільність алгоритму.

6. (Опційно) Копіювання: Скопіювати відсортовані елементи з `B` назад в `A`.

Псевдокод сортування підрахунком

```
function CountingSort(array A)
  n = length(A)
  if n <= 1
    return // Масив вже відсортований
  end if

  // Крок 1: Знаходження діапазону значень
  max_val = A[0]
  min_val = A[0]
  for i from 1 to n-1
    if A[i] > max_val
      max_val = A[i]
    end if
    if A[i] < min_val
      min_val = A[i]
    end if
  end for
```

```
  range_size = max_val - min_val + 1

  // Крок 2: Створення та ініціалізація масиву лічильників
  array count[range_size] // Ініціалізований нулями

  // Крок 3: Підрахунок частот
  for i from 0 to n-1
    count[A[i] - min_val] = count[A[i] - min_val] + 1
  end for

  // Крок 4: Обчислення кумулятивних сум
  for i from 1 to range_size - 1
    count[i] = count[i] + count[i-1]
  end for

  // Крок 5: Формування результуючого масиву
  array B[n] // Вихідний масив
  // Проходимо по вхідному масиву у зворотному порядку для стійкості
  for i from n-1 down to 0
    // Знаходимо позицію в масиві B
    position = count[A[i] - min_val] - 1
    B[position] = A[i]
    // Зменшуємо лічильник для наступного такого ж елемента
    count[A[i] - min_val] = count[A[i] - min_val] - 1
  end for

  // Крок 6: Копіювання результату назад в A
  for i from 0 to n-1
    A[i] = B[i]
  end for
end function
```

- `min_val`, `max_val`: Мінімальне та максимальне значення в масиві `A`.

- `range_size` (`k`): Розмір діапазону значень.

- `count`: Допоміжний масив для підрахунку частот та кумулятивних сум.

- `B`: Вихідний масив для збереження відсортованого результату.

Приклад роботи

Розглянемо сортування масиву `A = [4, 2, 2, 8, 3, 3, 1]`.

1. Діапазон: `min = 1`, `max = 8`. Розмір діапазону `k = 8 - 1 + 1 = 8`.

2. Масив лічильників: `count` розміром 8, ініціалізований `[0, 0, 0, 0, 0, 0, 0, 0]`.

3. Підрахунок частот:

- `count[1-1]++` (для `1`), `count[2-1]++` (для `2`), `count[2-1]++` (для `2`), ...

- Результат: `count = [1, 2, 2, 1, 0, 0, 0, 1]`

- *Пояснення:* ***`count[0]`****=1 (одна '1'),* ***`count[1]`****=2 (дві '2'),* ***`count[2]`****=2 (дві '3'), etc.*

4. Кумулятивні суми:

- `count[1] = count[1] + count[0] = 2 + 1 = 3`

- `count[2] = count[2] + count[1] = 2 + 3 = 5`

- `count[3] = count[3] + count[2] = 1 + 5 = 6`

- ...

- Результат: `count = [1, 3, 5, 6, 6, 6, 6, 7]`

- *Пояснення:* ***`count[1]`****=3 означає, що є 3 елементи, менші або рівні 2.* ***`count[7]`****=7 означає, що є 7 елементів, менших або рівних 8.*

5. Формування результуючого масиву B:

- `i=6`: `A[6]=1`. Позиція = `count[1-1]-1` = `count[0]-1` = `1-1=0`. `B[0]=1`. `count[0]` стає `0`.

- `i=5`: `A[5]=3`. Позиція = `count[3-1]-1` = `count[2]-1` = `5-1=4`. `B[4]=3`. `count[2]` стає `4`.

- `i=4`: `A[4]=3`. Позиція = `count[3-1]-1` = `count[2]-1` = `4-1=3`. `B[3]=3`. `count[2]` стає `3`.

- `i=3`: `A[3]=8`. Позиція = `count[8-1]-1` = `count[7]-1` = `7-1=6`. `B[6]=8`. `count[7]` стає `6`.

- ...і так далі.

- Фінальний B: `[1, 2, 2, 3, 3, 4, 8]` (Примітка: для повноти, елемент '4' з індексу 0 буде поставлений на позицію `count[4-1]-1 = count[3]-1 = 6-1=5`, `B[5]=4`).

6. Масив `A` оновлюється значеннями з `B`.

Аналіз складності

- Часова складність: O(n + k).

- де `n` — кількість елементів у масиві, `k` — діапазон значень (`max - min + 1`).

- Кожен з кроків (пошук діапазону, підрахунок, кумуляція, побудова) виконується за час, пропорційний `n` або `k`.

- Алгоритм є лінійним, якщо `k` є порівнянним з `n` (тобто, `k = O(n)`). Якщо `k` значно більше за `n` (напр., $k = n^2$), то складність визначається `k` і алгоритм стає неефективним.

- Просторова складність: O(n + k).

- `O(k)` для масиву лічильників `count`.

- `O(n)` для вихідного масиву `B`.

Властивості алгоритму

- Стійкість: Алгоритм є стійким. Зворотний прохід на етапі формування вихідного масиву гарантує, що елементи з однаковими значеннями зберігають свій початковий відносний порядок.

- Не “in-place”: Вимагає значного обсягу додаткової пам’яті.

- Не порівняльний: Працює шляхом підрахунку та індексації, а не порівняння елементів.

Область застосування

- Сортування цілих чисел у малому діапазоні: Це основне і найефективніше застосування.

- Як підпрограма в інших алгоритмах: Сортування підрахунком є ключовим компонентом порозрядного сортування (Radix Sort), де воно використовується для стабільного сортування елементів за кожним розрядом.

- Обробка даних та гістограми: Логіка підрахунку частот може бути використана для побудови гістограм розподілу значень.

Переваги та недоліки

Переваги:

- Дуже висока швидкість: Лінійний час $O(n + k)$ є асимптотично швидшим за будь-який алгоритм сортування, що базується на порівняннях.

- Стійкість: Важлива властивість для складних сортувань.

Недоліки:

- Обмеженість застосування: Ефективний лише для цілих чисел (або даних, що легко відображаються в них).

- Високі вимоги до пам’яті: Просторова складність $O(n+k)$ робить його непрактичним, якщо діапазон значень `k` дуже великий (наприклад, для сортування 64-бітних чисел).

- Не “in-place”: Потребує створення копії масиву.

Контрольні запитання

1. У чому полягає основний принцип сортування підрахунком, який дозволяє йому працювати швидше за $O(n\log n)$?

2. Поясніть призначення кроку обчислення кумулятивних сум у масиві `count`. Що зберігає `count[j]` після цього кроку?

3. Чому для забезпечення стабільності алгоритму фінальний прохід по вхідному масиву потрібно робити у зворотному порядку?

4. Наведіть приклад даних, для яких сортування підрахунком буде вкрай неефективним.

5. Яка часова та просторова складність сортування `n` елементів у діапазоні від 0 до n^2?

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз продуктивності сортування підрахунком.

2. Мета:

- Реалізувати стабільну версію сортування підрахунком.

- Дослідити залежність продуктивності алгоритму від діапазону значень `k` відносно кількості елементів `n`.

- Застосувати алгоритм для сортування об’єктів за цілочисловим ключем.

3. Завдання:

- Реалізувати функцію `CountingSort`.

- Написати програму, яка тестує продуктивність алгоритму в таких умовах:

- `n` фіксоване, `k` зростає (напр., `k=n, k=n*10, k=n*100`).

- `k` фіксоване, `n` зростає.

- Створити клас/структуру (напр., “Студент” з полями “Ім’я” та “Оцінка”) і відсортувати масив таких об’єктів за полем “Оцінка”. Перевірити стабільність.

4. Опис: Лабораторна робота фокусується на розумінні сильних та слабких сторін сортування підрахунком. Студенти повинні на практиці побачити, як зростання діапазону `k` перетворює лінійний алгоритм на повільний та

вимогливий до пам’яті. Завдання на сортування об’єктів допоможе закріпити розуміння стабільності.

5. Порядок виконання:

- Реалізувати функцію `CountingSort`.

- Написати програму-бенчмарк, що генерує масиви з заданими `n` та `k` і вимірює час сортування.

- Провести серію експериментів, змінюючи `n` та `k`, і занести результати в таблицю.

- Реалізувати сортування для масиву об’єктів, додавши в масив об’єкти з однаковими ключами, щоб перевірити, чи зберігається їхній відносний порядок.

- Зробити висновки щодо практичного застосування алгоритму.

6. Результат:

- Програма, що реалізує стабільне сортування підрахунком.

- Звіт з аналізом продуктивності, що демонструє залежність від `n` та `k`.

- Демонстрація стабільності алгоритму.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
    - Функція `CountingSort`.
    - Приклад сортування масиву об’єктів.
- Продемонструвати роботу додатку:
    - Сортування масиву цілих чисел.
    - Порівняння часу роботи для малого та великого діапазону `k`.
- Бути готовим відповісти на питання:
    - Яка складність алгоритму і чому вона записується як $O(n + k)$?
    - Чому цей алгоритм не є універсальним?
    - Як забезпечується стабільність у вашій реалізації?
    - Де сортування підрахунком використовується як частина іншого, більш складного алгоритму?

Тести

1. Яка часова складність сортування підрахунком?

1) O(n log n)

2) O(n)

3) O(n + k), де k - діапазон значень

4) O(k)

2. Сортування підрахунком є ефективним, коли:

1) Діапазон значень k значно більший за кількість елементів n.

2) Елементи є числами з плаваючою комою.

3) Кількість елементів n дуже мала.

4) Діапазон значень k порівнянний з кількістю елементів n.

3. Яка з властивостей НЕ притаманна стабільному сортуванню підрахунком?

1) Стабільність.

2) Лінійна часова складність.

3) Сортування “на місці” (in-place).

4) Не використовує порівняння елементів.

4. Який крок в алгоритмі стабільного сортування підрахунком є ключовим для визначення фінальних позицій елементів?

1) Підрахунок частот кожного елемента.

2) Пошук мінімального та максимального значень.

3) Копіювання результату в початковий масив.

4) Обчислення кумулятивних сум у масиві лічильників.

5. Для сортування 1 мільйона 32-бітних цілих чисел, сортування підрахунком є:

1) Ідеальним вибором.

2) Гарним вибором, але повільнішим за Quicksort.

3) Вкрай непрактичним через величезні вимоги до пам’яті.

4) Неможливим для застосування.

Відповіді:

1. c)

2. d)

3. c)

4. d)

5. c)

Тести підвищеної складності

1. Якщо вхідний масив `A=[2, 5, 2, 3, 5]` (`min=2, max=5`), яким буде масив `count` після кроку обчислення кумулятивних сум?

1) `[2, 1, 0, 2]`

2) `[2, 3, 3, 5]`

3) `[2, 3, 5]`

4) `[0, 2, 1, 2]`

2. Чому для забезпечення стабільності фінальний цикл побудови вихідного масиву `B` іде у зворотному порядку (з `n-1` до `0`)?

1) Це швидше для процесора.

2) Це гарантує, що останній з однакових елементів вхідного масиву буде розміщений на останній з доступних для нього позицій.

3) Це єдиний спосіб правильно використовувати кумулятивні суми.

4) Це не обов’язково, прямий порядок теж працюватиме.

3. Сортування підрахунком найчастіше використовується як стабільна підпрограма для:

1) Quicksort.

2) Bubble Sort.

3) Merge Sort.

4) Radix Sort.

4. Якщо з реалізації сортування підрахунком прибрати крок обчислення кумулятивних сум і просто заповнювати вихідний масив на основі частот, що ми втратимо?

1) Лінійну складність.

2) Здатність сортувати числа.

3) Стійкість та здатність сортувати об’єкти за ключем.

4) Нічого, алгоритм працюватиме так само.

5. Яка просторова складність сортування `n` цілих чисел в діапазоні `[-m, +m]`?

1) O(n + m)

2) O(n + 2m)

3) O(n)

4) O(m)

6. Що станеться, якщо спробувати відсортувати масив `[0.5, 0.2, 0.9]` за допомогою сортування підрахунком?

1) Алгоритм відсортує їх коректно.

2) Алгоритм неможливо застосувати, оскільки він працює лише з цілими числами.

3) Алгоритм дасть помилку під час виконання.

4) Алгоритм відсортує їх як `[0, 0, 0]`.

7. Якщо масив `A` вже відсортований, чи працюватиме сортування підрахунком швидше?

1) Так, значно швидше, за час O(n).

2) Так, трохи швидше.

3) Ні, час роботи не зміниться і залишиться O(n+k).

4) Повільніше, через додаткові перевірки.

8. Якщо `k` (діапазон) є константою, яка асимптотична складність сортування підрахунком?

1) O(1)

2) O(n)

3) O(n log n)

4) O(n + log k)

9. Який крок алгоритму робить його не “in-place”?

1) Пошук min/max.

2) Створення масиву лічильників `count`.

3) Створення вихідного масиву `B`.

4) 2) та с).

10. Якщо прибрати зсув на `min_val` і працювати з індексами напряму, що потрібно гарантувати щодо вхідних даних?

1) Що всі числа унікальні.

2) Що всі числа є степенями двійки.

3) Що всі числа є невід'ємними.

4) Що масив відсортований.

Відвовіді:

1. 2) Частоти: `count=[2,1,0,2]` (для 2,3,4,5). Кумулятивні: `count[0]=2`, `count[1]=2+1=3`, `count[2]=3+0=3`, `count[3]=3+2=5`.

2. 2) Це гарантує, що останній з однакових елементів вхідного масиву буде розміщений на останній з доступних для нього позицій. Перший з однакових елементів буде оброблено останнім і поставлено на першу доступну позицію.

3. 4) Порозрядне сортування вимагає стабільного сортування на кожному розряді, і Counting Sort ідеально для цього підходить.

4. 3) Стійкість та здатність сортувати об'єкти за ключем. Проста версія може сортувати лише самі числа, але не може правильно розставити об'єкти з цими числовими ключами.

5. 2) Діапазон `k = m - (-m) + 1 = 2m + 1`. Тому складність O(n + k) = O(n + 2m + 1), що спрощується до O(n + 2m).

6. 2) Алгоритм неможливо застосувати, оскільки він працює лише з цілими числами. Він покладається на те, що значення можна використати як індекси масиву.

7. 3). Алгоритм не є адаптивним.

8. 2) Якщо `k` є константою, то складність $O(n + k)$ перетворюється на $O(n)$.

9. 4) Обидва масиви, `count` та `B`, вимагають додаткової пам'яті.

10. 3) Якщо є від'ємні числа, спроба доступу до `count[-x]` призведе до помилки виходу за межі масиву.

Завдання для самостійної роботи

1. Реалізуйте "наївну" версію сортування підрахунком (без кумулятивних сум) і поясніть, чому вона не є стабільною.

2. Модифікуйте сортування підрахунком для обробки масивів, що містять від'ємні числа.

3. Напишіть програму, яка візуалізує вміст масиву `count` на кожному з етапів алгоритму.

4. Реалізуйте сортування підрахунком для сортування рядків, що складаються з одного символу (наприклад, `['c', 'a', 'b']`).

5. Дослідіть, як можна було б адаптувати сортування підрахунком для невеликого діапазону чисел з плаваючою комою (напр., з двома знаками після коми).

6. Проаналізуйте накладні витрати на пам'ять. Напишіть функцію, що розраховує, скільки додаткової пам'яті знадобиться для сортування заданого масиву.

7. Порівняйте на практиці продуктивність вашої реалізації з вбудованим сортуванням вашої мови програмування на масивах з малим діапазоном `k`.

8. Напишіть тести, які перевіряють стабільність вашої реалізації сортування підрахунком.

9. Дослідіть, як змінюється продуктивність, якщо замість створення вихідного масиву `B` спробувати модифікувати масив `A` "на місці" (це можливо, але складніше).

10. Реалізуйте сортування підрахунком, використовуючи хеш-таблицю замість масиву `count`. Які переваги та недоліки такого підходу?

Теми лабораторних робіт

1. Реалізація та аналіз залежності продуктивності `Counting Sort` від співвідношення `n` та `k`.

2. Розробка стабільного `Counting Sort` для сортування складних об'єктів за цілочисловим ключем.

3. Порівняльний аналіз `Counting Sort` та `Radix Sort` для сортування 32-бітних цілих чисел.

4. Використання `Counting Sort` для вирішення задач на підрахунок частот та побудову гістограм.

5. Дослідження та реалізація `Counting Sort` для обробки від'ємних чисел та розширених наборів символів.

6. Аналіз просторової складності: оптимізація використання пам'яті в `Counting Sort`.

7. Створення візуалізатора, що демонструє роботу `Counting Sort` по кроках.

8. Розробка гібридного сортування, що обирає між **`QuickSort`** та **`Counting Sort`** залежно від діапазону даних.

9. Застосування **`Counting Sort`** для попередньої обробки даних перед використанням інших алгоритмів.

10. Аналіз продуктивності **`Counting Sort`** в умовах обмеженої пам'яті.

9.4. Гібридні та просунуті сортування: найкраще з усіх світів

Ми дослідили класичні алгоритми — прості, ефективні та навіть лінійні. Кожен з них має свої сильні сторони та слабкості. Quicksort надзвичайно швидкий у середньому, але має ризик деградації до $O(n^2)$. Merge Sort стабільний і надійний, але вимагає додаткової пам'яті. Heapsort гарантує $O(n\log n)$ на місці, але повільніший на практиці. Insertion Sort неперевершений на малих або майже відсортованих масивах.

Що, якби можна було взяти найкращі риси кожного і створити алгоритм, який був би швидким, надійним і адаптивним одночасно? Саме цю ідею реалізують **гібридні та просунуті алгоритми сортування**. Вони не винаходять нову фундаментальну парадигму, а розумно комбінують існуючі, щоб досягти оптимальної продуктивності в реальних умовах.

У цьому розділі ми зазирнемо під капот сучасних сортувальних машин:

- Timsort: Прагматичний чемпіон, що лежить в основі сортування в Python та Java. Це витончений гібрид Merge Sort та Insertion Sort, розроблений для блискучої роботи на реальних даних, які часто містять вже впорядковані послідовності.

- Introsort (інтроспективне сортування): Гарантія швидкості та надійності, що є стандартом у C++. Він починає з надшвидкого Quicksort, але пильно "спостерігає" за глибиною рекурсії. Якщо вона стає завеликою (що свідчить про ризик $O(n^2)$), він миттєво перемикається на Heapsort, гарантуючи найгірший випадок $O(n\log n)$.

- Smoothsort: Елегантний та теоретично красивий нащадок Heapsort. Цей алгоритм модифікує структуру піраміди так, щоб досягти лінійної складності $O(n)$ на вже відсортованих даних, чого не може класичний Heapsort, зберігаючи при цьому гарантію $O(n\log n)$.

Ці алгоритми — вершина еволюції сортування, де глибокі теоретичні знання поєднуються з інженерною мудрістю для створення інструментів, якими ми користуємося щодня.

9.4.1 Timsort: Прагматичне сортування для реального світу

Timsort — це високопродуктивний, стабільний, гібридний алгоритм сортування, створений Тімом Петерсом у 2002 році для мови програмування Python. Його головна мета — досягти оптимальної продуктивності на реальних наборах даних, які, на відміну від випадкових, часто містять частково впорядковані послідовності.

Алгоритм настільки ефективний, що став стандартом у Python, Java (для об'єктів), Android та інших системах. Timsort геніально поєднує найкращі риси сортування вставкою та сортування злиттям.

Основна ідея:

Timsort працює у дві фази:

1. Пошук та підготовка відсортованих підмасивів (ранів): На першому етапі масив розбивається на відсортовані частини.

2. Інтелектуальне злиття ранів: На другому етапі ці частини ефективно зливаються до отримання повністю відсортованого масиву.

Фаза 1: Пошук та створення "ранів"

Алгоритм не ділить масив наосліп. Він шукає "природні рани" — неперервні послідовності елементів, що вже відсортовані за зростанням або за спаданням.

- Якщо ран відсортований за спаданням, Timsort просто перевертає його, роблячи зростаючим.

- `minrun`: Для ефективного злиття рани не повинні бути надто короткими. Timsort обчислює мінімальний розмір рану (`minrun`), зазвичай від 32 до 65. Якщо знайдений природний ран коротший за `minrun`, він розширюється за допомогою бінарного сортування вставкою.

- Бінарне сортування вставкою: Це варіація звичайного сортування вставкою, яка використовує бінарний пошук для знаходження місця вставки нового елемента. Це зменшує кількість порівнянь, що робить його швидшим для розширення ранів.

Фаза 2: Стратегія злиття за допомогою стеку

Це серце Timsort. Замість простого рекурсивного злиття, Timsort використовує стек для зберігання ранів та зливає їх за розумною політикою.

- Кожен знайдений ран (його індекс та довжина) додається на стек.

- Політика злиття: Після додавання нового рану, алгоритм перевіряє три верхні рани на стеку (назвемо їх довжини X, Y, Z, де Z — на вершині). Якщо порушено один з інваріантів:

- `len(X) > len(Y) + len(Z)`

- `len(Y) > len(Z)` ...тоді виконується злиття. Зазвичай зливаються два менші з трьох верхніх ранів. Це гарантує, що рани, які зливаються, мають приблизно однаковий розмір, що є найефективнішим сценарієм для процедури злиття.

- Цей процес триває, доки всі рани не будуть об'єднані в один.

Оптимізація злиття: "Galloping Mode"

Якщо під час злиття Timsort помічає, що він послідовно бере багато елементів (наприклад, 7 і більше) з одного рану, він переходить у "режим галопу". Замість покрокового порівняння, він використовує експоненційний пошук, щоб швидко знайти, куди вставити елемент з іншого рану, дозволяючи копіювати великі блоки даних за один раз.

Повна реалізація (псевдокод)

Повна реалізація є складною, тому ми розіб'ємо її на логічні функції. Для ясності, `merge` показано у класичному вигляді без "галопу".

1. Допоміжні функції

```
const MIN_MERGE = 32

// Обчислення minrun
function calculate_minrun(n)
  r = 0 // Бітовий прапорець
  while n >= MIN_MERGE
    r = r | (n & 1)
    n = n >> 1
  end while
  return n + r
end function

// Бінарне сортування вставкою для малих ранів
function binary_insertion_sort(array A, int left, int right)
  for i from left + 1 to right
    key = A[i]
    // ... знайти місце для key в A[left...i-1] за допомогою бінарног
о пошуку ...
```

```
    pos = binary_search(A, key, left, i - 1)
    // ... зсунути елементи і вставити key ...
  end for
end function

// Процедура злиття (класична)
function merge(array A, run1_start, run1_len, run2_start, run2_len)
  // ... створити тимчасові масиви для двох ранів ...
  // ... злити їх назад в масив A, зберігаючи стабільність ...
end function
```

2. Основний алгоритм Timsort

```
function Timsort(array A)
  n = length(A)
  if n < 2
    return
  end if

  minrun = calculate_minrun(n)

  runs_stack = new empty_stack()
  pos = 0

  // Фаза 1: Пошук, створення та початкове злиття ранів
  while pos < n
    // Знаходимо або створюємо наступний ран
    run_start = pos
    // Рахуємо природний ран
    run_end = find_natural_run(A, run_start)

    // Якщо ран спадний, перевертаємо його
    if is_descending(A, run_start, run_end)
      reverse(A, run_start, run_end)
    end if

    run_len = run_end - run_start + 1

    // Якщо ран коротший за minrun, розширюємо його
    if run_len < minrun
      new_len = min(minrun, n - pos)
      binary_insertion_sort(A, pos, pos + new_len - 1)
      run_len = new_len
    end if

    // Додаємо ран на стек
```

```
    push(runs_stack, (run_start, run_len))

    // Зливаємо рани згідно з політикою
    merge_collapse(A, runs_stack)

    pos = pos + run_len
  end while

  // Фаза 2: Фінальне злиття всіх ранів, що залишились
  merge_force_collapse(A, runs_stack)
end function

// Політика злиття
function merge_collapse(array A, stack s)
  while size(s) > 1
    // Позначаємо три верхні рани Z, Y, X
    Z = top(s)
    Y = second(s)
    X = third(s) // Якщо існує

    // Перевіряємо інваріанти
    if size(s) >= 3 and X.len <= Y.len + Z.len
      if X.len < Z.len
        merge(A, from X and Y)
      else
        merge(A, from Y and Z)
      end if
    else if Y.len <= Z.len
      merge(A, from Y and Z)
    else
      break // Інваріанти задоволені
    end if
  end while
end function

// Примусове злиття в кінці
function merge_force_collapse(array A, stack s)
  while size(s) > 1
    Y = top(s)
    Z = second(s)
    merge(A, from Y and Z)
  end while
end function
```

Аналіз складності

- Часова складність: $O(n\log n)$ у середньому та найгіршому випадках, $O(n)$ у найкращому.

- Просторова складність: $O(k)$ або $O(\log n)$ у кращих випадках, $O(n)$ у найгіршому (де k - кількість ранів).

Властивості: Стабільний, адаптивний, гібридний.

Область застосування

- Стандартне сортування: Є алгоритмом за замовчуванням у Python, Java (для об'єктів), Android.

- Частково впорядковані дані: Надзвичайно ефективний для даних, які вже мають певну структуру, що часто трапляється на практиці (наприклад, дані з часовими мітками, об'єднання кількох наборів даних).

Переваги та недоліки

Переваги:

- Висока продуктивність на реальних даних.

- Стабільність та гарантована складність $O(n\log n)$.

- Ефективне використання пам'яті порівняно з класичним Merge Sort.

Недоліки:

- Висока складність реалізації: Алгоритм містить багато оптимізацій і його важко реалізувати з нуля.

- Накладні витрати: На повністю випадкових даних може бути трохи повільнішим за Quicksort.

Контрольні запитання

1. Чому Timsort вважається гібридним алгоритмом? Які алгоритми він поєднує?

2. Поясніть роль `minrun` та бінарного сортування вставкою у першій фазі алгоритму.

3. Що таке політика злиття і чому вона важлива для ефективності Timsort?

4. Що таке "режим галопу" і коли він активується?

5. Чому Timsort, незважаючи на складність, став стандартом у таких мовах, як Python та Java?

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз ключових компонентів алгоритму Timsort.

2. Мета:

- Зрозуміти будівельні блоки, з яких складається Timsort.

- Реалізувати функції пошуку ранів та бінарного сортування вставкою.

- Оцінити ефективність цих компонентів на різних типах даних.

3. Завдання:

- Написати функцію, що знаходить “природні рани” в масиві (зростаючі та спадні).

- Реалізувати бінарне сортування вставкою і порівняти його швидкість (за кількістю порівнянь) зі звичайним сортуванням вставкою.

- Створити набори тестових даних, що імітують реальні ситуації (майже відсортовані, два об’єднані відсортовані списки).

- Проаналізувати, як компоненти Timsort працюють на цих даних.

4. Опис: Повна реалізація Timsort є надто складною для стандартної лабораторної роботи. Робота фокусується на розумінні його основних механізмів. Студенти реалізують частини, які роблять Timsort адаптивним, і досліджують їхню поведінку.

5. Порядок виконання:

- Реалізувати функцію `find_natural_run(A, start)`, яка повертає кінець рану.

- Реалізувати функцію `binary_insertion_sort(A, left, right)`.

- Створити тестові масиви.

- Запустити `find_natural_run` на цих масивах і вивести знайдені рани.

- Порівняти кількість порівнянь для `binary_insertion_sort` та звичайного `insertion_sort` на невеликих масивах.

- Зробити висновки про роль кожного компонента в загальній ефективності Timsort.

6. Результат:

- Програма, що реалізує ключові компоненти Timsort.

- Звіт з аналізом роботи цих компонентів.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:

    - Функція пошуку природних ранів.

- Функція бінарного сортування вставкою.

- Продемонструвати роботу додатку:
    - Пошук ранів на масиві з частковою впорядкованістю.
- Бути готовим відповісти на питання:
    - З яких двох основних алгоритмів складається Timsort?
    - Яка роль `minrun`?
    - Що таке політика злиття і чому вона важлива?

Тести

1. Timsort є гібридним алгоритмом, що поєднує:

1) Quicksort та Heapsort.

2) Bubble Sort та Selection Sort.

3) Insertion Sort та Merge Sort.

4) Radix Sort та Counting Sort.

2. Яка часова складність Timsort у найкращому випадку (наприклад, на вже відсортованому масиві)?

1) O(1)

2) O(n)

3) O(n log n)

4) $O(n^2)$

3. Що таке “природний ран” в контексті Timsort?

1) Будь-який підмасив розміром `minrun`.

2) Неперервна послідовність елементів, що вже є відсортованою (зростаючою або спадною).

3) Частина масиву, що обробляється одним рекурсивним викликом.

4) Масив після першого проходу Merge Sort.

4. Для чого Timsort використовує “режим галопу” (galloping mode)?

1) Щоб швидко знаходити опорний елемент.

2) Щоб пропускати великі блоки даних під час злиття, коли один ран значно “випереджає” інший.

3) Щоб перевертати спадні рани.

4) Щоб обчислювати `minrun`.

5. Ключова перевага Timsort, що зробила його стандартом для Python, це:

1) Простота реалізації.

2) Найкраща продуктивність на повністю випадкових даних.

3) Висока ефективність на реальних, частково впорядкованих даних.

4) Мінімальне використання пам’яті серед усіх сортувань.

Відповіді:

1. c)
2. b)
3. b)
4. b)
5. c)

Тести підвищеної складності

1. Мета політики злиття в Timsort — це:

1) Завжди зливати найбільші рани в першу чергу.

2) Уникати злиттів, доки всі рани не будуть знайдені.

3) Зливати рани приблизно однакового розміру, щоб зберегти збалансованість і ефективність.

4) Виконувати злиття у випадковому порядку.

2. Timsort використовує бінарне сортування вставкою, а не звичайне, тому що:

1) Воно швидше за часом у найгіршому випадку.

2) Воно зменшує кількість операцій обміну.

3) Воно зменшує кількість порівнянь, що особливо корисно при дорогих операціях порівняння.

4) Воно стабільне, на відміну від звичайного сортування вставкою.

3. Яка з властивостей НЕ є гарантованою для Timsort?

1) Стікість.

2) Часова складність O(n log n) у найгіршому випадку.

3) Просторова складність O(log n) у найгіршому випадку.

4) Часова складність O(n) у найкращому випадку.

4. Чим Timsort принципово відрізняється від Introsort?

1) Timsort є стійким, а Introsort — ні.

2) Timsort базується на Merge Sort, а Introsort — на Quicksort.

3) Timsort оптимізований для частково впорядкованих даних, а Introsort — для уникнення найгіршого випадку Quicksort.

4) Усі перелічені відповіді є вірними.

5. Якщо `minrun` для масиву розміром `n=1000` дорівнює 63, а Timsort знаходить природний ран довжиною 40, що він зробить?

1) Залишить цей ран як є і буде шукати наступний.

2) Проігнорує його і почне шукати заново.

3) Розширить його до 63 елементів, відсортувавши додаткові 23 елементи за допомогою бінарного сортування вставкою.

4) Розділить його на менші частини.

6. На масиві, що складається з двох відсортованих половин (наприклад, `[1, 5, 10, 20, 2, 6, 8, 15]`), як поведе себе Timsort?

1) Розділить його рекурсивно, як звичайний Merge Sort.

2) Знайде два великі рани і виконає одне фінальне злиття.

3) Застосує сортування вставкою до всього масиву.

4) Зазнає деградації до $O(n^2)$.

7. Наявність спадних ранів (напр., `[10, 8, 5, 2]`) для Timsort — це:

1) Найгірший випадок, що уповільнює алгоритм.

2) Не проблема, оскільки він просто перевертає їх і обробляє як звичайні рани.

3) Ситуація, яка змушує його перемикатися на Quicksort.

4) Причина для використання “режиму галопу”.

8. Який з компонентів Timsort найбільше сприяє його адаптивності?

1) Політика злиття на стеку.

2) Початковий етап пошуку природних ранів.

3) Використання `minrun`.

4) “Режим галопу”.

9. Яке твердження про просторову складність Timsort є найбільш точним?

1) Вона завжди O(n).

2) Вона завжди O(1).

3) Вона адаптивна і залежить від структури даних, варіюючись від O(log n) до O(n).

4) Вона завжди O(log n).

10. Якщо порівнювати Timsort з класичним Merge Sort на повністю випадкових даних, то Timsort, швидше за все, буде:

1) Значно швидшим.

2) Значно повільнішим.

3) Приблизно таким же за швидкістю, можливо, трохи повільнішим через додаткові накладні витрати.

4) Не зможе відсортувати масив.

Відповіді:

1. 3) Зливати рани приблизно однакового розміру, щоб зберегти збалансованість і ефективність. Це запобігає неефективним злиттям дуже великого рану з дуже малим.

2. 3) Воно зменшує кількість порівнянь, що особливо корисно при дорогих операціях порівняння. Кількість зсувів залишається такою ж, як у звичайного сортування вставкою.

3. 3) Просторова складність O(log n) у найгіршому випадку. У найгіршому випадку Timsort може потребувати до O(n) додаткової пам’яті.

4. 4) Усі перелічені відповіді є вірними. Це два різні за філософією гібридні алгоритми, що вирішують різні проблеми.

5. 3) Розширить його до 63 елементів, відсортувавши додаткові 23 елементи за допомогою бінарного сортування вставкою. Це гарантує, що всі рани мають мінімальний розмір `minrun`.

6. 2) Знайде два великі рани і виконає одне фінальне злиття. Це ідеальний сценарій, що демонструє силу алгоритму.

7. 2) Не проблема, оскільки він просто перевертає їх і обробляє як звичайні рани. Це частина етапу пошуку та підготовки ранів.

8. 2) Початковий етап пошуку природних ранів. Саме здатність розпізнавати та використовувати існуючий порядок є основою його адаптивності.

9. 3) Вона адаптивна і залежить від структури даних, варіюючись від O(log n) до O(n).

10. 3) Приблизно таким же за швидкістю, можливо, трохи повільнішим через додаткові накладні витрати. На повністю хаотичних даних переваги Timsort не проявляються.

Завдання для самостійної роботи

1. Реалізуйте функцію обчислення `minrun` для заданого `n` згідно з логікою Timsort.

2. Напишіть функцію, яка приймає масив і повертає список знайдених в ньому природних ранів (їхні початки та довжини).

3. Реалізуйте бінарне сортування вставкою.

4. Напишіть програму, яка імітує роботу стеку злиття в Timsort: отримуючи на вхід послідовність ранів, вона має застосовувати політику злиття і показувати, які рани і коли зливаються.

5. Дослідіть, як працює експоненційний пошук, що лежить в основі “режиму галопу”.

6. Порівняйте на практиці продуктивність вбудованого сортування вашої мови (Python/Java) з класичним Merge Sort на різних типах даних.

7. Створіть набір даних, який буде “найгіршим випадком” для Timsort (тобто, змусить його використати $O(n)$ додаткової пам’яті).

8. Поясніть, чому стабільність є важливою вимогою для Timsort.

9. Проаналізуйте вихідний код реалізації Timsort у відкритій бібліотеці (наприклад, CPython).

10. Напишіть есе, що порівнює філософії дизайну Timsort та Introsort.

Теми лабораторних робіт

1. Реалізація та аналіз ключових компонентів Timsort: `minrun`, пошук ранів, бінарне сортування вставкою.

2. Створення спрощеної версії Timsort (без “галопу” та складної політики злиття) та порівняння її з Merge Sort.

3. Візуалізатор роботи Timsort: демонстрація пошуку ранів та процесу злиття на стеку.

4. Аналіз продуктивності Timsort на даних з різним ступенем впорядкованості.

5. Дослідження “режиму галопу”: реалізація та оцінка приросту продуктивності під час злиття.

6. Порівняльний аналіз Timsort та Introsort: філософія, реалізація, продуктивність.

7. Оптимізація використання пам’яті в Timsort.

8. Адаптація Timsort для сортування зв’язних списків.

9. Застосування принципів Timsort для розробки власного гібридного алгоритму сортування.

10. Аналіз впливу розміру `minrun` на загальну продуктивність Timsort.

9.5. Пошук k-ої порядкової статистики

У попередніх розділах ми зосередилися на алгоритмах, що впорядковують усі елементи масиву. Але що, якщо нам не потрібен весь відсортований набір? Уявіть, що вам потрібно знайти медіанну зарплату в компанії з тисячами співробітників або 95-й перцентиль часу відгуку сервера. Сортувати весь масив даних лише для того, щоб знайти одне-єдине значення, — це неефективно.

Саме для таких завдань існують алгоритми пошуку порядкової статистики (selection algorithms). Це спеціалізована група методів, покликаних знаходити k-й за величиною елемент у невпорядкованому масиві, роблячи це значно швидше, ніж повне сортування.

1. Що таке порядкова статистика?

Порядкова статистика — це елемент, який би стояв на k-й позиції, якби масив був повністю відсортований за зростанням. Інакше кажучи, це k-й найменший елемент.

Ранг `k` може приймати значення від 1 до `n`, де `n` — кількість елементів у масиві.

Класичні приклади:

- **Мінімум**: Це 1-ша порядкова статистика (`k=1`).

- **Максимум**: Це n-та порядкова статистика (k=n).

- **Медіана**: Це "серединний" елемент.

- Для масиву з непарною кількістю елементів n медіана — це унікальний елемент на позиції $k = (n + 1)/2$. - Для масиву з парною n існує дві медіани: нижня ($k = n/2$) та верхня ($k = n/2 + 1$). Зазвичай під медіаною розуміють нижню.

Квантилі (Quantiles): Це загальніший термін, що описує точки, які ділять відсортований набір даних на рівні за розміром групи.

- **Квартилі** (Quartiles): Ділять дані на чотири частини.

- Q1 (перший квартиль): 25-й перцентиль, відсікає нижні 25% даних.

- Q2 (другий квартиль): Медіана (50-й перцентиль).

- Q3 (третій квартиль): 75-й перцентиль, відсікає нижні 75% даних.

Перцентилі (Percentiles): Ділять дані на 100 частин.

Наприклад, знаходження 99-го перцентиля часу завантаження веб-сторінки дозволяє ігнорувати екстремальні викиди, але зрозуміти, з якою максимальною затримкою стикаються 99% користувачів.

Основна перевага алгоритмів селекції полягає в тому, що вони дозволяють знайти будь-яку з цих статистик за очікуваний лінійний час $O(n)$, що значно швидше за сортування ($O(n\log n)$).

2. Ключова ідея: модифікований "Розділяй і володарюй"

В основі ефективних алгоритмів селекції лежить та сама геніальна ідея, що і в Quicksort — розділення масиву (`partition`).

1. Вибір опорного елементу: Обирається опорний елемент.

2. Розділення: Масив перевпорядковується так, що всі елементи, менші за опору, знаходяться зліва, а більші або рівні — справа. Після цього кроку опора опиняється на деякій позиції p.

3. Аналіз позиції:

- Якщо p == k, то ми знайшли шуканий елемент!

- Якщо k < p, то k-й елемент точно знаходиться у лівому підмасиві.

- Якщо k > p, то k-й елемент знаходиться у правому підмасиві.

Ключова відмінність від Quicksort полягає в тому, що ми робимо лише один рекурсивний виклик — для тієї частини масиву, де гарантовано знаходиться

наш елемент. Це кардинально знижує складність. Якщо Quicksort описується рекурентним співвідношенням $T(n) = 2T(n/2) + O(n)$, що дає $O(n\log n)$, то алгоритми селекції описуються як $T(n) = T(n/2) + O(n)$, що, згідно з Основною теоремою про рекурентні співвідношення, дає $O(n)$.

3. QuickSelect: Практичний вибір

QuickSelect — це пряма реалізація описаної вище ідеї.

- Ідея: Використовувати випадковий опорний елемент, виконати розділення і рекурсивно викликати себе для однієї з половин.

- Середня складність: $O(n)$. Випадковий вибір опори з високою ймовірністю дає збалансоване розділення.

- Найгірший випадок: $O(n^2)$. Якщо нам систематично не щастить і опорний елемент виявляється крайнім, розмір масиву зменшується лише на одиницю за крок, що призводить до квадратичної складності. Хоча на практиці це малоймовірно, така можливість існує.

- Оптимізації: Для покращення продуктивності та зменшення ймовірності найгіршого випадку використовуються ті ж техніки, що і в Quicksort, наприклад, “медіана трьох” для вибору опори.

4. Median-of-Medians: Гарантована лінійність

Median-of-Medians — це детермінований алгоритм, який гарантує лінійну часову складність $O(n)$ навіть у найгіршому випадку. Його мета — знайти “достатньо хороший” опорний елемент, який гарантовано відсікає значну частину масиву на кожному кроці.

Алгоритм вибору опори:

1. Розбити масив на групи по 5 елементів.

2. Знайти медіану в кожній п’ятірці (це можна зробити за константний час, наприклад, за допомогою сортування вставкою).

3. Створити новий масив з цих медіан.

4. Рекурсивно викликати алгоритм Median-of-Medians для цього нового масиву, щоб знайти його медіану.

5. Ця “медіана медіан” і буде нашим опорним елементом.

Математично доведено, що такий опорний елемент гарантовано більший за щонайменше 30% елементів і менший за щонайменше 30% елементів. Це запобігає вкрай незбалансованому розділенню і забезпечує лінійну складність.

Недоліки: Алгоритм значно складніший у реалізації, а більша константа в $O(n)$ робить його на практиці повільнішим за QuickSelect для середньостатистичних даних.

5. Практичні рекомендації

- QuickSelect — ваш вибір у 99% випадків. Він простий, швидкий, а ймовірність найгіршого випадку на реальних даних мізерно мала.

- Median-of-Medians — використовується, коли гарантії продуктивності є критично важливими. Наприклад, у системах реального часу або в алгоритмах, де селекція є підпрограмою, і її деградація до $O(n^2)$ може зламати часову складність основного алгоритму.

У наступних секціях ми детально розглянемо псевдокоди, аналіз просторової та часової складності та наведемо приклади застосування цих потужних алгоритмів.

9.5.1 QuickSelect: Швидкий пошук за лінійний час

QuickSelect — це алгоритм пошуку k-ої порядкової статистики, розроблений Тоні Хоаром, автором Quicksort. Цей алгоритм є яскравим прикладом того, як можна адаптувати ідею одного потужного алгоритму (сортування) для вирішення іншої, більш вузької задачі (пошуку) з вражаючою ефективністю. Замість того, щоб сортувати весь масив, QuickSelect знаходить k-й найменший елемент за очікуваний лінійний час $O(n)$.

Основний принцип роботи: “Розділяй, але не володарюй над усім”

QuickSelect використовує ту ж саму процедуру `partition` (розділення), що і Quicksort, але з однією ключовою відмінністю в логіці.

1. Вибір опорного елемента: З масиву обирається опорний елемент.

2. Розділення: Масив перевпорядковується так, що всі елементи, менші за опору, знаходяться зліва від неї, а всі, що більші або рівні, — справа. Після цього опора опиняється на своїй фінальній відсортованій позиції з індексом `p`.

3. Аналіз та рекурсія: На відміну від Quicksort, який рекурсивно викликається для *обох* частин, QuickSelect аналізує позицію опори `p` і продовжує пошук лише в одній з частин:

- Якщо `p == k`, то опорний елемент і є шуканим. Алгоритм завершується.

- Якщо `k < p`, то k-й елемент точно знаходиться у лівому підмасиві. Рекурсивно шукаємо k-й елемент у діапазоні `[left ... p-1]`.

- Якщо `k > p`, то k-й елемент знаходиться у правому підмасиві. Рекурсивно шукаємо `k`-й елемент у діапазоні `[p+1 ... right]`.

Саме відмова від другого рекурсивного виклику і є тим секретом, що знижує середню часову складність з $O(n \log n)$ до $O(n)$.

Схема розділення Ломуто в QuickSelect

Для простоти та наочності в QuickSelect часто використовують схему розділення Ломуто, оскільки вона гарантує, що опорний елемент одразу стає на свою фінальну позицію, що спрощує логіку рекурсивних викликів.

Псевдокод процедури `LomutoPartition`:

```
function LomutoPartition(array A, int left, int right)
  pivot_value = A[right] // Обираємо останній елемент як опору
  store_index = left

  for i from left to right - 1
    if A[i] < pivot_value
      swap(A[store_index], A[i])
      store_index = store_index + 1
  end for

  swap(A[store_index], A[right])
  return store_index // Повертаємо фінальний індекс опори
```

Псевдокод алгоритму QuickSelect

`k` тут — це k-й найменший елемент, тому для 0-індексованого масиву ми шукатимемо індекс `k-1`.

```
// A - масив, left, right - межі, k - порядкова статистика (1-індексо
вана)
function QuickSelect(array A, int left, int right, int k)

  // Поки підмасив містить більше одного елемента
  while left <= right
    // Базовий випадок: якщо шукаємо в масиві з одного елемента
    if left == right
      return A[left]
    end if

    // Виконуємо розділення. pivot_index - це фінальна позиція опори
    pivot_index = LomutoPartition(A, left, right)
```

```
    // k_index - це 0-індексована позиція, яку ми шукаємо
    k_index = k - 1

    // Три випадки:
    if k_index == pivot_index
      // Знайшли!
      return A[k_index]
    else if k_index < pivot_index
      // Шукаємо в лівій частині
      right = pivot_index - 1
    else
      // Шукаємо в правій частині
      // k не змінюється, бо ми шукаємо k-ту позицію в оригінальному
масиві
      left = pivot_index + 1
    end if
  end while
end function
```

Примітка: наведена ітеративна версія, щоб уникнути глибокої рекурсії. Рекурсивна реалізація виглядає аналогічно, але з рекурсивними викликами замість оновлення ***`left`*** *та* ***`right`****. Да і швидкість ітеративної версії краще.*

Приклад роботи

Знайдемо 4-й найменший елемент (`k=4`) у масиві `A = [7, 2, 8, 1, 5, 9, 3]`. Ми шукаємо елемент на індексі `k-1 = 3`.

1. **Ітерація 1:** `left=0, right=6`. `pivot=3`.

- `partition` для `[7, 2, 8, 1, 5, 9, **3**]` дає: `[2, 1, **3**, 8, 5, 9, 7]`.

- `pivot_index = 2`.

- Ми шукаємо індекс `3`. Оскільки `3 > 2`, продовжуємо пошук у правому підмасиві: `[8, 5, 9, 7]`. `left` стає `3`.

2. **Ітерація 2:** `left=3, right=6`. Підмасив `[8, 5, 9, 7]`. `pivot=7`.

- `partition` для `[8, 5, 9, **7**]` дає: `[5, **7**, 9, 8]`. (Елементи в масиві `A` на позиціях `[3..6]`).

- `pivot_index = 4` (в оригінальному масиві).

- Ми шукаємо індекс `3`. Оскільки `3 < 4`, продовжуємо пошук у лівому підмасиві: `[5]`. `right` стає `3`.

3. **Ітерація 3:** `left=3, right=3`. Підмасив `[5]`.

- `left == right`. Це базовий випадок. Повертаємо `A[3]`.

- **Результат: 5**.

Перевірка: відсортований масив `[1, 2, 3, **5**, 7, 8, 9]`. 4-й елемент — це `5`.

Аналіз складності

- Часова складність:

- Середній випадок: O(n). Якщо розділення збалансоване, на кожному кроці ми відкидаємо приблизно половину масиву. Робота, виконана на кожному кроці: $n + n/2 + n/4 + \cdots + 1 = 2n$. Таким чином, складність є лінійною.

- Найгірший випадок: $O(n^2)$. Виникає, якщо опорний елемент систематично обирається найменшим або найбільшим. Тоді розмір масиву зменшується лише на 1 за крок.

- Просторова складність: O(log n).

- Для рекурсивної версії потрібна пам'ять для стеку викликів, яка в середньому випадку становить $O(\log n)$, а в найгіршому — $O(n)$. Ітеративна версія має складність $O(1)$.

Рандомізація для уникнення найгіршого випадку Щоб практично усунути ризик $O(n^2)$, перед розділенням обирають випадковий елемент і міняють його місцями з останнім. Це робить виникнення найгіршого випадку вкрай малоймовірним.

Область застосування

QuickSelect є надзвичайно корисним, коли потрібне одне конкретне значення з ранжованого набору, а не весь відсортований масив.

- Пошук квантилів: знаходження медіани, перцентилів, квартилів у великих наборах даних.

- Бази даних: для виконання запитів типу `SELECT ... ORDER BY ... LIMIT k`, щоб знайти `k` найбільших/найменших значень.

- Обробка зображень: знаходження медіанного значення кольору для медіанного фільтра.

- Комп'ютерна графіка: визначення межі об'єктів (bounding box).

Переваги та недоліки

Переваги:

- Висока середня швидкість: Очікуваний лінійний час $O(n)$ є асимптотично оптимальним.

- Сортування “на місці” (in-place): Модифікує вхідний масив, не потребуючи значної додаткової пам’яті.

- Простота реалізації порівняно з детермінованим аналогом Median-of-Medians.

Недоліки:

- Найгірший випадок $O(n^2)$: Хоча рідкісний при рандомізації, теоретичний ризик існує.

- Нестабільність: Не зберігає відносний порядок однакових елементів.

- Модифікує вхідний масив: Якщо початковий порядок елементів важливий, потрібно працювати з копією.

Контрольні запитання

1. У чому полягає ключова відмінність у рекурсивних викликах між Quicksort та QuickSelect, і як це впливає на їхню часову складність?

2. Поясніть, чому середній час роботи QuickSelect є лінійним ($O(n)$).

3. Що спричиняє найгірший випадок $O(n^2)$ в QuickSelect, і як рандомізація допомагає його уникнути?

4. Чи є QuickSelect стабільним алгоритмом? Чому?

5. Наведіть приклад задачі, де використання QuickSelect значно ефективніше, ніж повне сортування масиву.

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз продуктивності алгоритму QuickSelect.

2. Мета:

- Реалізувати алгоритм QuickSelect.

- Порівняти його практичну продуктивність з методом повного сортування.

- Дослідити вплив рандомізації на уникнення найгіршого випадку.

3. Завдання:

- Реалізувати функцію `QuickSelect` з використанням схеми розділення Ломуто.

- Написати програму, яка знаходить k-й елемент двома способами:

1. За допомогою вашої функції `QuickSelect`.

2. Шляхом повного сортування масиву (напр., вбудованою функцією) та взяття елемента з індексу `k-1`.

- Порівняти час виконання обох методів на великих масивах.

- Реалізувати рандомізований вибір опорного елемента і протестувати його на відсортованому масиві, щоб показати, як він уникає $O(n^2)$.

4. Опис: Лабораторна робота фокусується на практичній демонстрації переваг QuickSelect. Студенти повинні реалізувати алгоритм і наочно побачити на бенчмарках, наскільки він швидший за повне сортування для задачі пошуку порядкової статистики. Також робота покаже важливість правильного вибору опорного елемента.

5. Порядок виконання:

- Реалізувати функцію `LomutoPartition`.

- На її основі реалізувати ітеративну або рекурсивну функцію `QuickSelect`.

- Написати головну програму, що генерує великий масив випадкових чисел.

- Запустити пошук медіани (k = n/2) за допомогою `QuickSelect` та повного сортування, вимірявши час для обох.

- Створити відсортований масив і запустити на ньому `QuickSelect` з простим вибором опори (останній елемент), щоб спровокувати найгірший випадок.

- Додати рандомізацію вибору опори і повторити тест на відсортованому масиві.

- Проаналізувати результати.

6. Результат:

- Програма, що реалізує QuickSelect.

- Звіт з порівнянням часу виконання та демонстрацією ефективності рандомізації.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
    - Процедура `partition`.
    - Основна функція `QuickSelect`.
    - Рандомізований вибір опори.

- Продемонструвати роботу додатку:
    - Порівняння швидкості QuickSelect та повного сортування.
    - Демонстрація найгіршого випадку та його уникнення.
- Бути готовим відповісти на питання:
    - Яка складність QuickSelect і чому?
    - Чому QuickSelect модифікує вхідний масив?
    - У чому різниця між пошуком k-го елемента та пошуком елемента зі значенням k?

Тести

1. Яка середня часова складність алгоритму QuickSelect?

1) O(1)

2) O(n)

3) O(n log n)

4) $O(n^2)$

2. QuickSelect є модифікацією якого алгоритму?

1) Merge Sort.

2) Heapsort.

3) Quicksort.

4) Insertion Sort.

3. Найгірший випадок для QuickSelect можна ефективно уникнути за допомогою:

1) Використання схеми розділення Ломуто.

2) Рандомізованого вибору опорного елемента.

3) Збільшення розміру масиву.

4) Використання ітеративної версії.

4. Основна відмінність QuickSelect від Quicksort полягає у:

1) Способі вибору опорного елемента.

2) Використанні іншої процедури розділення.

3) Кількісті рекурсивних викликів на кожному кроці (один проти двох).

4) QuickSelect не використовує рекурсію.

5. Для якої задачі QuickSelect НЕ є підходящим інструментом?

1) Знаходження медіани масиву.

2) Перевірка, чи всі елементи масиву унікальні.

3) Знаходження 95-го перцентиля.

4) Знаходження 10-го найбільшого елемента.

Відповіді:

1. b)
2. c)
3. b)
4. c)
5. b)

Тести підвищеної складності

1. Якщо ми шукаємо k-й елемент (`k=5`) у масиві з 10 елементів, і після першого розділення `pivot` опинився на індексі `p=3` (0-індексація), де ми продовжимо пошук?

1) У лівому підмасиві, шукаючи 5-й елемент.

2) У правому підмасиві, шукаючи 5-й елемент.

3) У правому підмасиві, шукаючи (5-3-1)=1-й елемент.

4) Алгоритм завершено, результат `A[3]`.

2. Який алгоритм гарантує лінійний час $O(n)$ у найгіршому випадку для пошуку k-ої статистики?

1) QuickSelect з рандомізацією.

2) Iterative QuickSelect.

3) Median-of-Medians.

4) Heapsort.

3. Просторова складність $O(n)$ у рекурсивного QuickSelect виникає, коли:

1) Глибина рекурсії досягає $O(n)$.

2) Масив містить багато дублікатів.

3) Використовується схема Хоара.

4) Це неможливо, складність завжди $O(\log n)$.

4. Якщо потрібно знайти 5 найбільших елементів у великому масиві, який підхід буде найефективнішим?

1) Повністю відсортувати масив і взяти останні 5.

2) 5 разів запустити QuickSelect для знаходження n-го, (n-1)-го, ..., (n-4)-го елементів.

3) Запустити QuickSelect один раз для знаходження (n-4)-го елемента, а потім знайти 5 найбільших у правому підмасиві.

4) Використати мін-піраміду розміром 5.

5. Після виконання QuickSelect для пошуку медіани масиву `A`, що можна сказати про стан масиву?

1) Він буде повністю відсортований.

2) Елемент медіани буде на своїй правильній позиції, а всі елементи зліва будуть менші, а справа — більші.

3) Тільки елемент медіани буде на своїй позиції, порядок інших не гарантується.

4) Масив залишиться незмінним.

6. В ітеративній версії QuickSelect, яка структура даних неявно замінює рекурсивний стек?

1) Черга.

2) Піраміда.

3) Цикл `while` та змінні `left`/`right`.

4) Хеш-таблиця.

7. Чому QuickSelect зазвичай швидший за Median-of-Medians на практиці?

1) Median-of-Medians має гіршу асимптотичну складність.

2) QuickSelect використовує менше пам'яті.

3) Процедура вибору опори в Median-of-Medians має велику константу, що робить її повільною.

4) QuickSelect є стабільним.

8. Якщо `k` є дуже малим (напр., `k=1`), QuickSelect:

1) Працюватиме значно швидше, ніж для `k=n/2`.

2) Працюватиме значно повільніше.

3) Час роботи не залежить від значення `k`.

4) Завершиться за один крок.

9. Що станеться, якщо у рекурсивному виклику для правого підмасиву (`k_index > pivot_index`) забути змінити `k`?

1) Нічого, алгоритм працюватиме коректно.

2) Алгоритм буде шукати неправильний елемент у підмасиві і поверне хибний результат.

3) Алгоритм зациклиться.

4) Це призведе до помилки компіляції.

10. QuickSelect модифікує масив “на місці”. Це означає, що:

1) Він не використовує жодної додаткової пам’яті.

2) Він використовує $O(1)$ додаткової пам’яті, не враховуючи стек викликів.

3) Він створює копію масиву перед роботою.

4) Він працює лише з масивами, що зберігаються в кеші.

Відповіді:

1. 2) Індекс `k` не змінюється, бо він стосується оригінального масиву. Ми просто звужуємо діапазон пошуку.

2. 3) Це його головне призначення — гарантувати лінійний час.

3. 1) Це відбувається у найгіршому випадку незбалансованого розділення.

4. 4) Пройти по масиву, додаючи елементи в піраміду і видаляючи мінімум, якщо розмір перевищив 5. Час $O(n\log 5)$, що є $O(n)$. Варіант (c) також добрий, але (d) є класичним рішенням.

5. 2) Елемент медіани буде на своїй правильній позиції, а всі елементи зліва будуть менші, а справа — більші. Це прямий наслідок роботи процедури `partition`.

6. 3) Ітеративна версія симулює рекурсію, звужуючи межі в циклі.

7. 3) Процедура вибору опори в Median-of-Medians має велику константу, що робить її повільною. QuickSelect має менші накладні витрати на кожному кроці.

8. 1) Працюватиме значно швидше, ніж для `k=n/2`. З високою ймовірністю випадкова опора буде більшою за `k`-й елемент, і масив для рекурсії буде швидко зменшуватися.

9. 2) Алгоритм буде шукати неправильний елемент у підмасиві і поверне хибний результат. Це поширена помилка при реалізації.

10. 2) Він використовує $O(1)$ додаткової пам'яті, не враховуючи стек викликів. Це стандартне визначення "in-place" для рекурсивних алгоритмів.

Завдання для самостійної роботи

1. Реалізуйте рекурсивну версію QuickSelect.

2. Модифікуйте вашу реалізацію для використання схеми розділення Хоара.

3. Напишіть функцію, яка знаходить `m` найбільших елементів у масиві, використовуючи QuickSelect.

4. Порівняйте на практиці час роботи QuickSelect та Median-of-Medians (можна знайти реалізацію онлайн).

5. Реалізуйте QuickSelect для пошуку k-го елемента у зв'язному списку. Проаналізуйте його ефективність.

6. Напишіть програму, яка візуалізує процес розділення та вибору підмасиву в QuickSelect.

7. Дослідіть, як QuickSelect поводиться на масивах з великою кількістю дублікатів, і реалізуйте 3-way partition для покращення.

8. Напишіть тести, які перевіряють коректність роботи вашого QuickSelect на граничних випадках (k=1, k=n, порожній масив).

9. Реалізуйте функцію для знаходження медіани, яка обробляє випадки парної та непарної кількості елементів.

10. Дослідіть алгоритм Introselect, який є гібридом QuickSelect та Median-of-Medians.

Теми лабораторних робіт

1. Реалізація та порівняльний аналіз QuickSelect та Median-of-Medians.

2. Розробка утиліти для швидкого обчислення довільних квантилів (медіани, перцентилів) у великих файлах даних.

3. Порівняння ефективності QuickSelect та методів на основі пірамід для знаходження "Top-K" елементів.

4. Аналіз впливу різних стратегій вибору опорного елемента на продуктивність QuickSelect.

5. Розробка візуалізатора, що демонструє роботу QuickSelect та Quicksort на однакових даних.

6. Реалізація та тестування 3-way QuickSelect для даних з великою кількістю дублікатів.

7. Застосування QuickSelect для реалізації медіанного фільтра для обробки зображень.

8. Аналіз та реалізація Introselect для гарантованої лінійної продуктивності.

9. Порівняння ітеративної та рекурсивної реалізацій QuickSelect: аналіз продуктивності та використання пам’яті.

10. Розробка бібліотеки для статистичного аналізу, де QuickSelect є ключовим компонентом.

Розділ 10: Алгоритми на рядках та стиснення даних

У цьому розділі ми вивчимо дві взаємопов'язані дисципліни, що дозволяють ефективно обробляти, аналізувати та зберігати цю інформацію.

Спочатку ми зануримося в алгоритми на рядках.

Тут ми розглянемо класичні задачі, такі як миттєвий пошук підрядка в тексті за допомогою елегантних методів Кнута-Морріса-Пратта та Боєра-Мура. Ми навчимося вимірювати "схожість" рядків за допомогою редакційної відстані, що є основою для систем перевірки орфографії та біоінформатики. Далі ми перейдемо до просунутих суфіксних структур — суфіксних дерев та масивів, які дозволяють вирішувати складні задачі аналізу тексту за лічені миті.

Друга частина розділу присвячена стисненню даних. В епоху великих даних здатність зменшувати їхній розмір без втрати інформації є критично важливою для ефективного зберігання та швидкої передачі. Ми розкриємо секрети відомих алгоритмів, таких як сімейство LZ (що лежить в основі форматів ZIP та PNG) та революційне перетворення Берроуза-Вілера (BWT), яке є серцем сучасних архіваторів. Ви побачите, що стиснення — це не магія, а витончене застосування алгоритмів для пошуку та усунення надлишковості в даних.

Цей розділ озброїть вас інструментами для роботи з однією з найпоширеніших форм даних, відкриваючи двері до розробки пошукових систем, текстових редакторів, біоінформатичних аналізаторів та ефективних систем передачі даних.

10.1. Суфіксні структури: суфіксний масив, суфіксне дерево, суфіксний автомат.

Теоретичні відомості

Суфіксні структури даних є фундаментальними інструментами для ефективної обробки рядків та розв'язання широкого спектру задач пошуку підрядків, порівняння послідовностей та аналізу тексту. Ці структури базуються на концепції суфікса - підрядка, що починається з деякої позиції і продовжується до кінця рядка.

Суфіксні структури -це просунуті структури даних, що дозволяють за лічені миті відповідати на складні запити про підрядки всередині тексту. Їхня магія полягає в тому, що вони індексують усі суфікси заданого рядка, а це, у свою чергу, дає нам інформацію про усі можливі підрядки.

Ключова ідея: Будь-який підрядок рядка `T` є префіксом одного з його суфіксів.

Наприклад, у рядку `T = "banana"` підрядок `"nan"` є префіксом суфікса `"nana"`. Отже, якщо ми маємо ефективний спосіб працювати з усіма суфіксами, ми можемо ефективно працювати з усіма підрядками.

Для рядка S довжиною n, суфікс, що починається з позиції i, позначається як S[i..n-1]. Наприклад, для рядка "banana" суфіксами є: "banana", "anana", "nana", "ana", "na", "a" та порожній рядок.

Основні суфіксні структури включають:

- **Суфіксний масив** - компактне представлення всіх суфіксів у відсортованому порядку

- **Суфіксне дерево** - дерево, що містить всі суфікси рядка як шляхи від кореня

- **Суфіксний автомат** - мінімальний детермінований скінченний автомат, що розпізнає всі суфікси

Суфіксний масив

Суфіксний масив SA для рядка S довжиною n - це масив цілих чисел, де SA[i] містить початкову позицію i-го суфікса в лексикографічному порядку. Іншими словами, якщо відсортувати всі суфікси рядка S, то SA[i] вказує на початок i-го суфікса в цьому відсортованому списку.

Додатково часто використовується масив LCP (Longest Common Prefix), де LCP[i] містить довжину найдовшого спільного префікса між суфіксами SA[i] та SA[i-1].

Приклад: `T = "banana$"`

1. **Виписуємо всі суфікси та їхні індекси:**

- 0: “banana$”

- 1: “anana$”

- 2: “nana$”

- 3: “ana$”

- 4: “na$”

- 5: “a$”

- 6: “$”

2. **Сортуємо їх в алфавітному порядку:**

- “$”

- “a$”

- “ana$”

- “anana$”

- “banana$”

- “na$”

- “nana$”

3. **Записуємо початкові індекси відсортованих суфіксів:**

- `SA = [6, 5, 3, 1, 0, 4, 2]`

- Це і є суфіксний масив. `SA[0]=6`, бо найменший суфікс (“$") починається з індексу 6. `SA[1]=5`, бо другий за алфавітом суфікс ("a$”) починається з індексу 5, і так далі.

Побудова суфіксного масиву:

- Наївний підхід ($O(n^2 \log n)$): Створити масив усіх суфіксів і відсортувати його стандартним сортуванням. Порівняння двох суфіксів може зайняти до $O(n)$ часу, що робить цей метод занадто повільним.

- Ефективні алгоритми ($O(n \log n)$ або $O(n)$): Існують значно складніші, але швидші алгоритми. Один з класичних підходів ($O(n \log^2 n)$ або $O(n \log n)$) базується на ідеї “подвоєння”:

1. Спочатку суфікси сортуються за першим символом.

2. Потім вони сортуються за першими двома символами, використовуючи результати попереднього кроку.

3. Потім — за першими чотирма, вісьмома і т.д., доки довжина порівняння не перевищить n. Існують і лінійні алгоритми (SA-IS), але вони дуже складні для реалізації.

Наївний алгоритм побудови суфіксного масиву

```
Алгоритм NaiveSuffixArray(S):
    Вхід:
        S - рядок довжиною n
    Вихід:
        SA - суфіксний масив

    n ← довжина(S)
    suffixes ← порожній список

    // Створюємо всі суфікси з їх початковими позиціями
    для i від 0 до n-1:
        suffix ← структура {
            index: i,
            suffix: S[i..n-1]
        }
        додати suffix до suffixes

    // Сортуємо суфікси лексикографічно
    відсортувати suffixes за полем suffix

    // Створюємо суфіксний масив
    SA ← новий масив[n]
    для i від 0 до n-1:
        SA[i] ← suffixes[i].index

    повернути SA
```

Алгоритм побудови за O(n log n) - метод подвоєння

```
Алгоритм SuffixArrayDoubling(S):
    Вхід:
        S - рядок довжиною n
    Вихід:
        SA - суфіксний масив

    n ← довжина(S)
    SA ← новий масив[n]
    rank ← новий масив[n]
    temp ← новий масив[n]

    // Ініціалізація: сортуємо за першим символом
    для i від 0 до n-1:
        SA[i] ← i
        rank[i] ← код_символу(S[i])

    k ← 1
    поки k < n:
        // Функція порівняння для сортування
        функція compare(i, j):
            якщо rank[i] ≠ rank[j]:
                повернути rank[i] < rank[j]
            ri ← rank[i + k] якщо i + k < n інакше -1
            rj ← rank[j + k] якщо j + k < n інакше -1
            повернути ri < rj

        // Сортуємо суфікси за першими 2k символами
        відсортувати SA використовуючи compare

        // Оновлюємо ранги
        temp[SA[0]] ← 0
        для i від 1 до n-1:
            якщо compare(SA[i-1], SA[i]) або compare(SA[i], SA[i-1]):
                temp[SA[i]] ← temp[SA[i-1]] + 1
            інакше:
                temp[SA[i]] ← temp[SA[i-1]]

        скопіювати temp в rank
        k ← k * 2

    повернути SA
```

Лінійний алгоритм побудови (DC3/Skew)

```
Алгоритм DC3(S):
    Вхід:
```

```
    S - рядок довжиною n
Вихід:
    SA - суфіксний масив

якщо n < 3:
    повернути NaiveSuffixArray(S)

// Крок 1: Розділення суфіксів на три групи
n0 ← (n + 2) / 3
n1 ← (n + 1) / 3
n2 ← n / 3
n02 ← n0 + n2

s12 ← новий масив[n02 + 3]
SA12 ← новий масив[n02 + 3]
s0 ← новий масив[n0]
SA0 ← новий масив[n0]

// Збираємо позиції i де i mod 3 ≠ 0
j ← 0
для i від 0 до n + (n0 - n1):
    якщо i mod 3 ≠ 0:
        s12[j] ← i
        j ← j + 1

// Крок 2: Сортуємо трійки
RadixSort(s12, SA12, S, n02, алфавіт)

// Крок 3: Перевіряємо унікальність
// ... (детальна реалізація)

// Крок 4: Рекурсивний виклик якщо потрібно
// ... (детальна реалізація)

// Крок 5: Злиття результатів
// ... (детальна реалізація)

повернути SA
```

Використання суфіксного масиву

Маючи суфіксний масив, ми можемо швидко перевірити, чи міститься зразок `P` довжиною `m` у тексті `T`. Оскільки масив `SA` відсортований, ми можемо використати бінарний пошук.

Псевдокод пошуку зразка:

```
function find_substring_with_SA(string T, array SA, string P)
  n = length(T)
  m = length(P)
  left = 0, right = n - 1

  while left <= right
    mid = left + (right - left) / 2
    suffix_start = SA[mid]
    // Порівнюємо P з префіксом суфікса, на який вказує SA[mid]
    comparison = compare(P, T.substring(suffix_start, suffix_start +
m))

    if comparison == 0 // Знайдено
      return true
    else if comparison < 0 // P менший, шукаємо в лівій половині
      right = mid - 1
    else // P більший, шукаємо в правій половині
      left = mid + 1
  end while

  return false // Не знайдено
end function
```

Обчислення масиву LCP

Масив LCP (Longest Common Prefix) Часто разом із суфіксним масивом будують масив **LCP**. **LCP[i]** зберігає довжину найдовшого спільного префікса між суфіксами, що стоять на позиціях **i** та **i-1** у суфіксному масиві. Цей масив можна побудувати за $O(n)$ (алгоритм Касаї) і він є ключовим для вирішення багатьох складних задач (наприклад, пошук найдовшого підрядка, що повторюється).

```
Алгоритм ComputeLCP(S, SA):
    Вхід:
        S - рядок довжиною n
        SA - суфіксний масив
    Вихід:
        LCP - масив найдовших спільних префіксів

    n ← довжина(S)
    LCP ← новий масив[n]
    rank ← новий масив[n]

    // Обчислюємо обернений масив (ранги)
```

```
    для i від 0 до n-1:
        rank[SA[i]] ← i

    h ← 0
    для i від 0 до n-1:
        якщо rank[i] > 0:
            j ← SA[rank[i] - 1]

            // Порівнюємо символи
            поки i + h < n та j + h < n та S[i + h] = S[j + h]:
                h ← h + 1

            LCP[rank[i]] ← h

            якщо h > 0:
                h ← h - 1

    повернути LCP
```

Суфіксне дерево

Суфіксне дерево — це більш потужна та швидка (для запитів) структура, ніж суфіксний масив, але вона значно складніша і вимагає більше пам'яті.

Визначення: Це стиснене дерево (trie), яке зберігає всі суфікси рядка `T`.

- Кожен шлях від кореня до листка відповідає одному суфіксу рядка `T`.

- Ребра дерева мітяться не окремими символами, а цілими підрядками.

- Усі внутрішні вузли (крім кореня) мають щонайменше двох нащадків.

Приклад: `T = "aba$"`

- Суфікси: “aba$", "ba$”, “a$", "$”.

- Дерево матиме 4 листки. Від кореня буде два ребра: одне з міткою “a”, інше з “b”. Вузол після ребра “a” матиме два нащадки: одне з міткою “ba$", інше з "$”.

Структура вузла суфіксного дерева

```
Структура SuffixTreeNode:
    children - словник дочірніх вузлів (символ → вузол)
    start - початкова позиція підрядка на ребрі
    end - кінцева позиція підрядка на ребрі (або посилання на глобаль
ну змінну для листків)
    suffixLink - суфіксне посилання
    parent - батьківський вузол
    suffixIndex - індекс суфікса (для листків)
```

Алгоритм Укконена для побудови суфіксного дерева

```
Алгоритм UkkonenSuffixTree(S):
    Вхід:
        S - рядок довжиною n
    Вихід:
        root - корінь суфіксного дерева

    n ← довжина(S)
    root ← новий SuffixTreeNode()
    root.suffixLink ← root

    activeNode ← root
    activeEdge ← -1
    activeLength ← 0
    remainingSuffixCount ← 0
    leafEnd ← -1

    для i від 0 до n-1:
        leafEnd ← i
        remainingSuffixCount ← remainingSuffixCount + 1
        lastNewNode ← null

        поки remainingSuffixCount > 0:
            якщо activeLength = 0:
                activeEdge ← i

            якщо activeNode.children не містить S[activeEdge]:
                // Створюємо новий листок
                leaf ← новий SuffixTreeNode()
                leaf.start ← i
                leaf.end ← посилання на leafEnd
                leaf.suffixIndex ← i - remainingSuffixCount + 1
                activeNode.children[S[activeEdge]] ← leaf

                якщо lastNewNode ≠ null:
                    lastNewNode.suffixLink ← activeNode
                    lastNewNode ← null
            інакше:
                // Переходимо по існуючому ребру
                next ← activeNode.children[S[activeEdge]]
                якщо walkDown(next):
                    продовжити цикл

                // Перевіряємо наступний символ
```

```
                якщо S[next.start + activeLength] = S[i]:
                    якщо lastNewNode ≠ null та activeNode ≠ root:
                        lastNewNode.suffixLink ← activeNode
                        lastNewNode ← null
                    activeLength ← activeLength + 1
                    break

                // Розділяємо ребро
                splitNode ← новий SuffixTreeNode()
                activeNode.children[S[activeEdge]] ← splitNode
                splitNode.start ← next.start
                splitNode.end ← next.start + activeLength - 1

                next.start ← next.start + activeLength
                splitNode.children[S[next.start]] ← next

                leaf ← новий SuffixTreeNode()
                leaf.start ← i
                leaf.end ← посилання на leafEnd
                leaf.suffixIndex ← i - remainingSuffixCount + 1
                splitNode.children[S[i]] ← leaf

                якщо lastNewNode ≠ null:
                    lastNewNode.suffixLink ← splitNode
                lastNewNode ← splitNode

            remainingSuffixCount ← remainingSuffixCount - 1
            якщо activeNode = root та activeLength > 0:
                activeLength ← activeLength - 1
                activeEdge ← i - remainingSuffixCount + 1
            інакше якщо activeNode ≠ root:
                activeNode ← activeNode.suffixLink

    повернути root
```

Пошук підрядка в суфіксному дереві

```
Алгоритм SearchInSuffixTree(root, pattern):
    Вхід:
        root - корінь суфіксного дерева
        pattern - шуканий підрядок
    Вихід:
        позиції входжень pattern

    node ← root
    i ← 0
```

```
    поки i < довжина(pattern):
        якщо node.children містить pattern[i]:
            child ← node.children[pattern[i]]
            edgeLength ← child.end - child.start + 1
            j ← 0

            // Порівнюємо символи на ребрі
            поки j < edgeLength та i < довжина(pattern):
                якщо S[child.start + j] ≠ pattern[i]:
                    повернути порожній список
                i ← i + 1
                j ← j + 1

            якщо i < довжина(pattern):
                node ← child
        інакше:
            повернути порожній список

    // Збираємо всі позиції з піддерева
    positions ← CollectLeafIndices(node)
    повернути positions
```

Суфіксний автомат

Суфіксний автомат — це найпотужніша і водночас найабстрактніша з трьох структур.

Суфіксний автомат (DAWG - Directed Acyclic Word Graph) для рядка S - це мінімальний детермінований скінченний автомат, що розпізнає всі суфікси S.

Він має такі властивості:

- Це орієнтований ациклічний граф (DAG).

- Кожен шлях від початкової вершини відповідає унікальному підрядку.

- На відміну від дерева, автомат може мати кілька шляхів, що ведуть в одну і ту ж вершину, що дозволяє йо.му бути значно компактнішим за суфіксне дерево.

- Кількість станів не перевищує 2n - 1

- Кількість переходів не перевищує 3n – 4

Існує елегантний онлайн-алгоритм, що будує автомат за час $O(n)$. Він послідовно додає символи рядка і розширює автомат, створюючи нові стани та переходи.

Використання суфіксного автомата:

- Пошук підрядка: Як і в суфіксному дереві, пошук зразка P — це просто прохід по автомату від початкового стану. Складність: $O(m)$.

- Підрахунок унікальних підрядків: Кількість унікальних підрядків дорівнює кількості унікальних шляхів у графі, яку можна знайти за $O(n)$ за допомогою динамічного програмування.

- Багато інших задач: Суфіксний автомат є надзвичайно потужним інструментом для вирішення десятків складних рядкових задач.

Структура стану суфіксного автомата

```
Структура SuffixAutomatonState:
    len - довжина найдовшого рядка в класі еквівалентності
    link - суфіксне посилання
    next - словник переходів (символ → стан)
    isTerminal - чи є стан термінальним
    firstPos - перша позиція входження
    endPosCnt - кількість позицій закінчення
```

Алгоритм побудови суфіксного автомата

```
Алгоритм BuildSuffixAutomaton(S):
    Вхід:
        S - рядок довжиною n
    Вихід:
        states - масив станів автомата

    states ← [новий SuffixAutomatonState()]
    states[0].len ← 0
    states[0].link ← -1
    last ← 0

    для c в S:
        // Створюємо новий стан для суфікса
        cur ← довжина(states)
        додати новий SuffixAutomatonState() до states
        states[cur].len ← states[last].len + 1
        states[cur].firstPos ← states[cur].len - 1

        // Додаємо переходи з попередніх станів
        p ← last
        поки p ≠ -1 та states[p].next не містить c:
            states[p].next[c] ← cur
            p ← states[p].link
```

```
        якщо p = -1:
            states[cur].link ← 0
        інакше:
            q ← states[p].next[c]
            якщо states[p].len + 1 = states[q].len:
                states[cur].link ← q
            інакше:
                // Клонуємо стан q
                clone ← довжина(states)
                додати новий SuffixAutomatonState() до states
                states[clone].len ← states[p].len + 1
                states[clone].next ← копія states[q].next
                states[clone].link ← states[q].link
                states[clone].firstPos ← states[q].firstPos

                поки p ≠ -1 та states[p].next[c] = q:
                    states[p].next[c] ← clone
                    p ← states[p].link

                states[q].link ← clone
                states[cur].link ← clone

        last ← cur

    // Позначаємо термінальні стани
    p ← last
    поки p ≠ -1:
        states[p].isTerminal ← true
        p ← states[p].link

    повернути states
```

Пошук підрядка в суфіксному автоматі

```
Алгоритм SearchInSuffixAutomaton(states, pattern):
    Вхід:
        states - масив станів автомата
        pattern - шуканий підрядок
    Вихід:
        true якщо pattern є підрядком, false інакше

    currentState ← 0

    для c в pattern:
        якщо states[currentState].next містить c:
            currentState ← states[currentState].next[c]
```

```
        інакше:
            повернути false

    повернути true
```

Підрахунок кількості входжень

```
Алгоритм CountOccurrences(states):
    // Попередня обробка: підрахунок endPosCnt
    для і від довжина(states)-1 до 0 з кроком -1:
        якщо states[i].isTerminal:
            states[i].endPosCnt ← 1
        для кожного переходу states[i].next[c] → j:
            states[i].endPosCnt ← states[i].endPosCnt + states[j].end
PosCnt

    функція getOccurrenceCount(pattern):
        state ← 0
        для c в pattern:
            якщо states[state].next містить c:
                state ← states[state].next[c]
            інакше:
                повернути 0
        повернути states[state].endPosCnt
```

Порівняння суфіксних структур

Характеристика	Суфіксний масив	Суфіксне дерево	Суфіксний автомат
Час побудови	$O(n\log n)$ або $O(n)$	$O(n)$	$O(n)$
Просторова складність	$O(n)$	$O(n)$	$O(n)$
Час пошуку підрядка	$O(m\log n)$	$O(m)$	$O(m)$
Складність реалізації	Середня	Висока	Дуже висока
Практичне використання	Найбільш популярний	Використовується	Нішевий / для змагань

Область застосування

Суфіксні структури широко використовуються в:

- Біоінформатика - пошук генів, порівняння геномів, виявлення повторів

- Обробка тексту - індексація документів, пошук патернів, стиснення

- Пошукові системи - швидкий пошук підрядків у великих текстових корпусах

- Компілятори - оптимізація коду, виявлення дублювання

- Обробка природної мови - морфологічний аналіз, виявлення n-грам

- Криптографія - аналіз періодичності, пошук патернів

- Музична інформатика - пошук мелодійних патернів

- Детекція плагіату - порівняння документів, пошук запозичень

Приклади застосування

Приклад 1: Пошук найдовшого повторюваного підрядка

```
Алгоритм LongestRepeatedSubstring(S):
    SA ← побудувати суфіксний масив для S
    LCP ← обчислити масив LCP

    maxLength ← 0
    position ← -1

    для і від 1 до довжина(S)-1:
        якщо LCP[i] > maxLength:
            maxLength ← LCP[i]
            position ← SA[i]

    повернути S[position..position+maxLength-1]
```

Приклад 2: Пошук всіх паліндромів

Використовуючи суфіксне дерево для рядка, можна ефективно знайти всі паліндроми.

Приклад 3: Стиснення LZ77

Суфіксні структури дозволяють швидко знаходити найдовші повтори для алгоритмів стиснення.

Переваги та недоліки

Суфіксний масив

Переваги: - Компактне представлення - лише O(n) цілих чисел - Простота реалізації базової версії - Ефективне кешування через послідовний доступ - Легко зберігати на диску - Підтримує бінарний пошук

Недоліки: - Потребує додаткові структури для складних запитів - Складність лінійних алгоритмів побудови - Менша функціональність порівняно з деревом

Суфіксне дерево

Переваги: - Найбільша функціональність - Лінійний час для більшості операцій - Природне представлення для багатьох задач - Підтримує онлайн-побудову

Недоліки: - Велике споживання пам’яті - Складна реалізація - Погана локальність кешу - Важко ефективно зберігати на диску

Суфіксний автомат

Переваги: - Мінімальне представлення - Ефективний для підрахунку входжень - Природний для розпізнавання патернів - Підтримує динамічні операції

Недоліки: - Залежність від розміру алфавіту - Складність розуміння та реалізації - Обмежена функціональність для деяких задач

Контрольні запитання

1. Що таке суфіксний масив і як він пов’язаний з лексикографічним порядком суфіксів?

2. Яка основна ідея алгоритму подвоєння для побудови суфіксного масиву?

3. Чим суфіксне дерево відрізняється від звичайного префіксного дерева?

4. Що таке суфіксне посилання і яка його роль в алгоритмі Укконена?

5. Як суфіксний автомат представляє всі підрядки заданого рядка?

6. Яка просторова складність кожної з трьох суфіксних структур?

7. Для яких задач найкраще підходить кожна з суфіксних структур?

8. Як використовувати масив LCP для пошуку найдовшого спільного підрядка?

9. Чому кількість станів суфіксного автомата не перевищує 2n-1?

10. Як перетворити суфіксний масив у суфіксне дерево?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження суфіксних структур даних

2. Мета

Вивчити принципи побудови та використання суфіксних структур, реалізувати основні алгоритми, дослідити їх ефективність для різних задач обробки рядків.

3. Завдання

- Реалізувати побудову суфіксного масиву методом подвоєння
- Реалізувати обчислення масиву LCP
- Створити спрощену версію суфіксного дерева
- Реалізувати базовий суфіксний автомат
- Розробити алгоритми пошуку підрядків для кожної структури
- Реалізувати пошук найдовшого повторюваного підрядка
- Провести порівняльний аналіз структур
- Візуалізувати суфіксні структури

4. Опис

Лабораторна робота охоплює реалізацію трьох основних суфіксних структур даних. Студенти мають зрозуміти принципи їх побудови, реалізувати базові алгоритми та дослідити практичне застосування для типових задач обробки рядків.

5. Порядок виконання

- Створити клас SuffixArray з методом buildDoubling
- Реалізувати метод computeLCP для масиву LCP
- Імплементувати клас SimpleSuffixTree з базовою функціональністю
- Створити клас SuffixAutomaton з методом build
- Реалізувати методи пошуку для кожної структури
- Створити метод findLongestRepeatedSubstring
- Розробити візуалізатор для суфіксних структур
- Провести тестування на різних типах даних
- Виконати порівняльний аналіз продуктивності

6. Результат

Програма повинна: - Коректно будувати всі три суфіксні структури - Виконувати базові операції пошуку та аналізу рядків - Візуалізувати структури для малих рядків - Демонструвати переваги кожної структури -

Показувати статистику використання пам’яті та часу - Працювати з текстами різних розмірів

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас SuffixArray з методами побудови та пошуку
- Клас SimpleSuffixTree з базовою реалізацією
- Клас SuffixAutomaton з повною функціональністю
- Методи для розв’язання типових задач
- Модуль візуалізації структур
- Модуль тестування та бенчмаркінгу

2. Продемонструвати роботу додатку:

- Побудова суфіксних структур для прикладів
- Візуалізація структур для коротких рядків
- Пошук підрядків різними методами
- Знаходження найдовшого повторюваного підрядка
- Порівняння часу побудови та пошуку
- Аналіз використання пам’яті

3. Бути готовим відповісти на питання:

- Як працює алгоритм подвоєння?
- Чому масив LCP можна обчислити за лінійний час?
- Як організовані переходи в суфіксному автоматі?
- Які переваги має кожна структура?
- Як вибрати оптимальну структуру для конкретної задачі?

Тести

1. Суфіксний масив для рядка “banana” починається з індексу:

1) 0

2) 5

3) 3

4) 1

2. Максимальна кількість вузлів у суфіксному дереві для рядка довжини n:

1) n

2) 2n

3) 2n - 1

4) n^2

3. Часова складність побудови суфіксного масиву методом подвоєння:

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) $O(n^3)$

4. Суфіксний автомат розпізнає:

1) Лише суфікси рядка

2) Всі підрядки рядка

3) Лише префікси рядка

4) Лише паліндроми

5. Масив LCP зберігає:

1) Довжини всіх суфіксів

2) Довжини спільних префіксів сусідніх суфіксів у відсортованому порядку

3) Позиції суфіксів

4) Кількість входжень кожного суфікса

Відповіді:

1. b)

2. c)

3. b)

4. b)

5. b)

Тести підвищеної складності

1. Для рядка довжини n, яка максимальна сумарна довжина всіх ребер суфіксного дерева?

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) $O(n^3)$

2. Скільки різних підрядків має рядок довжини n?

1) n

2) n(n+1)/2

3) n(n+1)/2 + 1

4) 2^n

3. Мінімальна кількість станів суфіксного автомата для рядка з n однакових символів:

1) n

2) n + 1

3) 2n - 1

4) 1

4. Яка структура найефективніша для пошуку k-го лексикографічно найменшого підрядка?

1) Суфіксний масив

2) Суфіксне дерево

3) Суфіксний автомат

4) Всі однаково ефективні

5. При використанні хешування для побудови суфіксного масиву, яка очікувана складність?

1) O(n)

2) O(n log n)

3) $O(n \log^2 n)$

4) $O(n^2)$

6. Скільки суфіксних посилань має суфіксне дерево для рядка довжини n?

1) n

2) n - 1

3) O(n)

4) 2n - 1

7. Для знаходження найдовшого спільного підрядка двох рядків найефективніше використовувати:

1) Два суфіксних масиви

2) Узагальнене суфіксне дерево

3) Два суфіксних автомати

4) Конкатенацію рядків з роздільником

8. Яка структура підтримує ефективне додавання символів в кінець рядка?

1) Суфіксний масив

2) Суфіксне дерево (алгоритм Укконена)

3) Суфіксний автомат

4) b та c

9. Простір для зберігання всіх суфіксних посилань у суфіксному автоматі:

1) O(n)

2) O(n log n)

3) O(n × |Σ|)

4) $O(n^2)$

10. При стисненні суфіксного дерева в суфіксний масив втрачається можливість:

1) Лексикографічного сортування

2) Пошуку підрядків за O(m)

3) Підрахунку входжень

4) Знаходження LCP

Відповіді:

1. c) $O(n^2)$ - сума довжин всіх суфіксів

2. c) n(n+1)/2 + 1 - включаючи порожній підрядок

3. b) n + 1 - початковий стан плюс n станів для префіксів

4. c) - Суфіксний автомат з попереднім підрахунком підрядків

5. b) O(n log n) - очікувана складність з хешуванням

6. c) O(n) - кількість внутрішніх вузлів

7. b) - Узагальнене суфіксне дерево найефективніше

8. d) - І дерево Укконена, і суфіксний автомат підтримують онлайн-побудову

9. a) O(n) - одне посилання на стан

10. b) - Втрачається можливість пошуку за лінійний час від довжини патерна

Завдання для самостійної роботи

1. Реалізуйте алгоритм DC3 для побудови суфіксного масиву за лінійний час.

2. Створіть узагальнене суфіксне дерево для множини рядків з ефективним пошуком спільних підрядків.

3. Розробіть стиснене представлення суфіксного дерева з використанням суфіксного масиву та LCP.

4. Імплементуйте динамічний суфіксний автомат з підтримкою видалення символів.

5. Створіть алгоритм пошуку всіх tandemних повторів використовуючи суфіксні структури.

6. Реалізуйте ефективний алгоритм для знаходження найдовшої спільної підпослідовності k рядків.

7. Розробіть паралельну версію побудови суфіксного масиву для багатоядерних систем.

8. Створіть алгоритм стиснення тексту на основі суфіксних структур.

9. Імплементуйте пошук приблизних входжень з використанням суфіксного дерева.

10. Розробіть візуальний редактор для інтерактивного дослідження суфіксних структур.

Теми лабораторних робіт

1. Розробка системи індексації геномних баз даних з використанням суфіксних масивів.

2. Створення інструменту для виявлення плагіату в програмному коді на основі суфіксних дерев.

3. Імплементація пошукової системи для великих текстових корпусів з суфіксними структурами.

4. Розробка компресора тексту з використанням суфіксного автомата для пошуку повторів.

5. Створення системи автодоповнення для текстових редакторів на базі суфіксних структур.

6. Реалізація біоінформатичного пакету для аналізу геномних послідовностей.

7. Розробка музичного аналізатора для пошуку повторюваних мотивів у MIDI-файлах.

8. Створення системи моніторингу логів з ефективним пошуком патернів.

9. Імплементація інструменту для оптимізації регулярних виразів через суфіксні автомати.

10. Розробка бібліотеки суфіксних структур з підтримкою Unicode та великих алфавітів.

10.2. Пошук підрядка

У розділі 10.1 ми переходимо до фундаментальної задачі обробки рядків — пошуку підрядка. Йдеться про те, щоб у великому тексті (текстовому документі, потоці даних, геномі тощо) знайти всі входження заданого шаблона (або множини шаблонів). Попри простоту формулювання, ефективна індексація та алгоритмічна оптимізація тут критично важливі для пошукових систем, біоінформатики, обробки природної мови й багатьох інших сфер.

Найпростіший, наївний алгоритм порівнює кожну позицію тексту зі всіма символами шаблона, даючи часову складність $O(nm)$ (де n — довжина тексту, m — довжина шаблона). Це неприйнятно повільно для великих даних.

Щоб досягти лінійного або близького до нього часу, розроблені спеціальні прийоми й евристики, які дозволяють «перестрибувати» непотрібні порівняння або швидко оновлювати хеші, або попередньо обчислювати інформацію про само́ шаблон. Далі ми коротко ознайомимося з основними підходами.

1. Наївний алгоритм

- Простий у реалізації, без попередньої обробки.

- Перебір усіх зсувів, $O(nm)$ у гіршому випадку.

- Добре підходить для коротких текстів або коротких шаблонів.

2. Кнута–Морріса–Пратта (КМР)

- Передзахідний крок: побудова префікс-функції (π-масиву) за $O(m)$.

- Основний пошук: з використанням π-масиву кожний символ тексту обробляється не більше одного разу, $O(n)$.

- Ідеально підходить для одного довгого шаблона в тексті.

3. Рабіна–Карпа

- Використовує хеш-функцію для вікон тексту довжини m.

- Обчислення rolling-хешу дає середню складність $O(n)$, гірший — $O(nm)$ при численних колізіях.

- Зручно для одночасного пошуку кількох шаблонів або «пошуку схожості».

4. Бойєра–Мура

- Спирається на дві евристики: «погляд із правого краю» (bad-character) і «хороший суфікс» (good-suffix).

- Дозволяє пропускати значні частини тексту, працюючи в середньому в $O(n/m)$ або навіть близько до $O(n)$.

- Ефективний на практиці для довгих шаблонів і великих алфавітів.

5. Aho–Corasick

- Побудова автомата (трі-структури з фолбек-посиланнями) для множини шаблонів за $O(\Sigma m_i)$.

- Пошук усіх шаблонів одночасно за $O(n +$ кількість входжень$)$.

- Ідеальний вибір, коли треба шукати десятки або сотні (і більше) ключових фраз одночасно.

6. Z-алгоритм

- Обчислює Z-масив за $O(n)$, який для кожної позиції показує довжину найдовшого префікса, що починається з цієї позиції.

- Використовується для пошуку одного шаблона в тексті шляхом конкатенації “шаблон + $ + текст” і аналізу Z-значень.

- Простий у реалізації й дає гарантію лінійного часу.

Усі розглянуті методи мають свої сильні та слабкі сторони. В подальших секціях ми детально розберемо кожний алгоритм: покажемо псевдокод, крок за кроком пояснимо структури даних і евристики, проаналізуємо

часові та просторові складності, а також наведемо приклади практичного застосування.

10.2.1 Наївний алгоритм пошуку підрядка

Задача пошуку підрядка є однією з фундаментальних в обробці текстових даних. Вона полягає у знаходженні всіх входжень одного рядка, що називається зразком, всередині іншого, більшого рядка, що називається текстом.

Наївний (або прямолінійний) алгоритм пошуку — це найбільш інтуїтивний та простий підхід до вирішення цієї задачі. Його ідея полягає в тому, щоб послідовно перевіряти кожну можливу початкову позицію в тексті, чи не починається з неї зразок.

Основний принцип роботи: “ковзне вікно”

Уявіть, що ми накладаємо зразок на текст, створюючи “вікно” розміром із зразок.

1. Початкове вирівнювання: Спочатку вікно вирівнюється за першим символом тексту (індекс 0).

2. Посимвольне порівняння: Алгоритм порівнює символи зразка з відповідними символами тексту всередині вікна, зліва направо.

- Якщо всі символи збігаються, це означає, що знайдено входження. Алгоритм фіксує цю позицію.

- Якщо на будь-якому кроці виникає розбіжність, порівняння для поточного вікна припиняється.

3. Зсув вікна: Незалежно від того, чи було знайдено збіг, чи ні, вікно зсувається на одну позицію праворуч по тексту.

4. Повторення: Кроки 2 і 3 повторюються, доки вікно не досягне кінця тексту, де подальші збіги вже неможливі.

Цей метод іноді називають “методом грубої сили” (brute-force), оскільки він перевіряє всі можливі варіанти без будь-яких інтелектуальних оптимізацій.

Псевдокод алгоритму

```
// Функція знаходить перше входження зразка P в тексті T
// і повертає його початковий індекс, або -1, якщо не знайдено.
function NaiveStringSearch(string T, string P)
  n = length(T) // Довжина тексту
```

```
  m = length(P) // Довжина зразка

  // Зовнішній цикл: ітерує по всіх можливих початкових позиціях ("зс
увах")
  // Зсув s може бути від 0 до n-m включно.
  for s from 0 to n - m

    // Внутрішній цикл: посимвольне порівняння зразка з підрядком тек
сту
    match = true
    for j from 0 to m - 1
      // Якщо символи не збігаються, перериваємо внутрішній цикл
      if T[s + j] != P[j]
        match = false
        break
      end if
    end for

    // Якщо внутрішній цикл завершився без розбіжностей, знайдено збі
г
    if match == true
      return s // Повертаємо початковий індекс входження
    end if

  end for

  return -1 // Якщо зовнішній цикл завершився, збігів не знайдено
end function
```

- `T`, `P`: Вхідні рядки — текст та зразок.
- `n`, `m`: Їхні довжини.
- `s`: Індекс у тексті `T`, з якого починається поточна перевірка (“зсув”).
- `j`: Індекс у зразку `P` для посимвольного порівняння.

Приклад роботи

Розглянемо пошук зразка `P = "ABC"` у тексті `T = "ABABDABC"`. `n = 8`, `m = 3`.

- **`s = 0`:**
- Порівнюємо `P` з `T[0...2] = "ABA"`.
- `T[0] == P[0]` (‘A’ == ‘A’). Збіг.
- `T[1] == P[1]` (‘B’ == ‘B’). Збіг.

- `T[2] != P[2]` (‘A’ != ‘C’). **Невдача.**

- **s = 1:**

- Порівнюємо `P` з `T[1...3] = "BAB"`.

- `T[1] != P[0]` (‘B’ != ‘A’). **Невдача.**

- **s = 2:**

- Порівнюємо `P` з `T[2...4] = "ABD"`.

- `T[2] != P[0]` (‘A’ != ‘A’). Збіг.

- `T[3] != P[1]` (‘B’ != ‘B’). Збіг.

- `T[4] != P[2]` (‘D’ != ‘C’). **Невдача.**

- **s = 3:**

- Порівнюємо `P` з `T[3...5] = "DAB"`.

- `T[3] != P[0]` (‘D’ != ‘A’). **Невдача.**

- **s = 4:**

- Порівнюємо `P` з `T[4...6] = "ABC"`.

- `T[4] == P[0]` (‘A’ == ‘A’). Збіг.

- `T[5] == P[1]` (‘B’ == ‘B’). Збіг.

- `T[6] == P[2]` (‘C’ == ‘C’). Збіг.

- **Успіх!** Знайдено входження на позиції `s = 4`. Функція повертає `4`.

Аналіз складності

- Часова складність:

- Найгірший випадок: O(m × (n - m + 1)), що спрощується до O(m·n).

- Такий випадок виникає, коли для кожного зсуву `s` внутрішній цикл виконує максимальну кількість порівнянь, перш ніж знайти розбіжність (або не знайти її).

- Класичний приклад: `Текст = "AAAAAAAAAAAAAAAAAB"`, `Зразок = "AAAB"`. На кожному кроці алгоритм буде робити `m-1` успішних порівнянь і лише на `m`-му зазнаватиме невдачі.

- Найкращий випадок: O(m).

- Якщо перше ж вікно дає збіг, алгоритм виконає `m` порівнянь і завершить роботу.

- Просторова складність: O(1).

- Алгоритм не потребує додаткової пам’яті, пропорційної розміру вхідних даних. Усі операції виконуються з використанням кількох змінних-лічильників.

Область застосування

Незважаючи на свою неефективність у найгіршому випадку, наївний алгоритм має свої ніші:

- Освітні цілі: Це базовий алгоритм, з якого починається вивчення рядкових пошуків. Він є еталоном для порівняння з більш просунутими методами.

- Пошук у коротких текстах або з короткими зразками: Коли `n` та `m` є невеликими, накладні витрати на попередню обробку, необхідну для складніших алгоритмів (як КМП), можуть бути невиправданими.

- Прототипування: Коли потрібно швидко реалізувати базову функціональність пошуку без оптимізацій.

- Тексти з великим алфавітом: Для текстів, де ймовірність випадкового збігу символів мала (наприклад, не бінарні дані), найгірший випадок трапляється рідко.

Переваги та недоліки

Переваги:

- Надзвичайна простота: Алгоритм дуже легко зрозуміти, реалізувати та налагодити.

- Відсутність фази попередньої обробки: Пошук починається негайно, що може бути перевагою для одноразових пошуків коротких зразків.

- Мінімальне використання пам’яті: Просторова складність $O(1)$.

Недоліки:

- Вкрай низька продуктивність у найгіршому випадку ($O(m \cdot n)$): Це робить його абсолютно непридатним для роботи з великими текстами, особливо в біоінформатиці або текстових редакторах.

- Надлишкова робота: Алгоритм “забуває” інформацію, отриману на попередніх кроках. Якщо частина зразка збіглася, ця інформація ніяк не використовується для прискорення наступних зсувів.

Контрольні запитання

1. Поясніть принцип “ковзного вікна”, який використовується в наївному алгоритмі.

2. Скільки всього можливих початкових позицій (зсувів) перевіряє алгоритм для тексту довжиною `n` та зразка довжиною `m`?

3. Наведіть власний приклад тексту та зразка, який призведе до найгіршого випадку часової складності.

4. Чому наївний алгоритм вважається неефективним з точки зору повторного використання інформації?

5. В якій ситуації наївний алгоритм може бути кращим вибором, ніж алгоритм Кнута-Морріса-Пратта?

Методичні вказівки до лабораторної роботи

1. Назва: Реалізація та аналіз продуктивності наївного алгоритму пошуку підрядка.

2. Мета:

- Реалізувати наївний алгоритм пошуку підрядка.

- Експериментально дослідити його часову складність, зокрема, продемонструвати найгірший випадок.

- Порівняти продуктивність реалізації з вбудованими функціями пошуку в рядках.

3. Завдання:

- Реалізувати функцію, що знаходить індекс першого входження зразка в тексті.

- Реалізувати функцію, що знаходить та повертає індекси всіх входжень.

- Написати програму-бенчмарк, яка вимірює час пошуку на різних даних:

- Випадкові рядки.

- Рядки, що провокують найгірший випадок (напр., `T="a...ab"`, `P="a...b"`).

- Порівняти час роботи вашої функції з методом `find()` (або аналогом) стандартної бібліотеки вашої мови програмування.

4. Опис: Лабораторна робота спрямована на закріплення розуміння базового, але важливого алгоритму. Студенти повинні не тільки реалізувати його, але й на практиці побачити його слабкі сторони, створивши умови для найгіршого сценарію. Порівняння з оптимізованими бібліотечними функціями покаже важливість просунутих алгоритмів.

5. Порядок виконання:

- Реалізувати функцію `NaiveStringSearch`.

- Створити генератори тестових даних (випадкових та “найгірших”).

- У головній програмі провести серію тестів для різних довжин `n` та `m`.

- Для кожного тесту виміряти час роботи вашої функції та стандартної функції пошуку.

- Результати звести в таблицю.

- Побудувати графік, що показує квадратичну залежність часу від `n` у найгіршому випадку.

- Зробити висновки.

6. Результат:

- Програма, що реалізує наївний пошук.

- Звіт з таблицями та графіками, що демонструють продуктивність та підтверджують теоретичні оцінки складності.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділена функція наївного пошуку.

- Продемонструвати роботу додатку:

    - Пошук у простому тексті.

    - Демонстрація різкого уповільнення роботи на “найгіршому” наборі даних.

    - Порівняння швидкості з вбудованою функцією.

- Бути готовим відповісти на питання:

    - Яка складність алгоритму і чому?

    - Наведіть приклад найгіршого випадку.

    - Чому стандартні бібліотечні функції працюють значно швидше?

Тести

1. Яка часова складність у найгіршому випадку для наївного алгоритму пошуку підрядка?

1) O(n + m)

2) O(n log m)

3) O(n * m)

4) O(n)

2. Основна перевага наївного алгоритму пошуку це:

1) Висока швидкість на всіх типах даних.

2) Простота реалізації та відсутність потреби в додатковій пам’яті.

3) Ефективна попередня обробка зразка.

4) Гарантований лінійний час роботи.

3. Для пошуку зразка `P="AAAA"` в тексті `T="AAAAAAAA"` скільки порівнянь символів буде виконано?

1) 4

2) 16

3) 20

4) 5

4. Що робить наївний алгоритм, коли знаходить розбіжність між символом тексту та зразка?

1) Завершує роботу.

2) Зсуває зразок на довжину збігу.

3) Зсуває зразок на одну позицію праворуч і починає порівняння заново.

4) Повертається на початок тексту.

5. Наївний алгоритм НЕ використовує:

1) Цикли.

2) Порівняння символів.

3) Інформацію про структуру зразка для оптимізації зсувів.

4) Індексацію масиву.

Відповіді:

1. c)
2. b)
3. c)
4. c)
5. c)

Тести підвищеної складності

1. Для пошуку зразка `P="ababc"` в тексті `T="abababc"`, на якій позиції (0-індексація) буде знайдено перше входження?

1) 0

2) 1

3) 2

4) Входжень немає.

2. Скільки всього порівнянь символів виконає наївний алгоритм для `T="abcabcabc"` та `P="abca"`?

1) 4

2) 8

3) 12

4) 10

3. Якщо довжина тексту `n=1000`, а довжина зразка `m=500`, скільки разів виконається зовнішній цикл (`for s ...`)?

1) 1000

2) 500

3) 501

4) 499

4. Як зміниться часова складність наївного алгоритму, якщо шукати зразок у бінарному рядку (алфавіт з 2 символів)?

1) Покращиться до O(n+m).

2) Погіршиться, оскільки найгірший випадок стає більш імовірним.

3) Не зміниться.

4) Покращиться до O(n log m).

5. Який з наведених алгоритмів є прямою оптимізацією наївного, що використовує інформацію про невдалі порівняння?

1) Алгоритм Рабіна-Карпа.

2) Алгоритм Кнута-Морріса-Пратта (КМП).

3) Сортування підрахунком.

4) Бінарний пошук.

6. Якщо потрібно знайти всі входження, а не тільки перше, як зміниться складність у найгіршому випадку?

1) Не зміниться, залишиться O(n*m).

2) Покращиться, бо ми робимо більше корисної роботи.

3) Погіршиться до $O(n^2m)$.

4) Покращиться до O(n log m).

7. Наївний алгоритм робить `n-m+1` ітерацій. На кожній ітерації він робить до `m` порівнянь. У якому випадку він зробить найменше порівнянь (окрім знаходження збігу на першому кроці)?

1) `T="abcdef"`, `P="xyz"`.

2) `T="aaaaaa"`, `P="aab"`.

3) `T="bbbbbb"`, `P="abb"`.

4) `T="ababab"`, `P="aba"`.

8. Просторова складність наївного алгоритму O(1) означає, що:

1) Він не використовує жодної змінної.

2) Кількість додаткової пам’яті не залежить від довжини тексту та зразка.

3) Він працює лише з одним символом за раз.

4) Він не може працювати з великими файлами.

9. Якщо зразок `P` має довжину `m=1`, яка буде часова складність пошуку в тексті довжиною `n`?

1) O(1)

2) O(n)

3) O(m*n)

4) O(log n)

10. Головна причина, чому наївний алгоритм “погано” працює на `T="aaaaa..."` та `P="aaab"`, полягає в тому, що:

1) Він не вміє працювати з однаковими символами.

2) Зсув на 1 позицію після невдачі є неефективним, бо алгоритм не враховує, що `m-1` символів вже збіглися.

3) Алгоритм не може обробляти рядки, довші за розмір кешу.

4) Використання двох циклів є вродженим недоліком.

Відповіді:

1. 3) Позиція 0: “ababa” != “ababc”. Позиція 1: “babab” != “ababc”. Позиція 2: “ababc” == “ababc”.

2. 3) Всього: 4+1+1+4+1+1 = 12. Відповідь 3)

3. 3) Цикл `for s from 0 to n - m`. `1000 - 500 = 500`. Цикл від 0 до 500 включно, що становить 501 ітерацію.

4. 2) Погіршиться, оскільки найгірший випадок стає більш імовірним. Часті збіги префіксів призводять до більшої кількості порівнянь на кожному зсуві.

5. 2) Алгоритм Кнута-Морріса-Пратта (КМП). Він будує префікс-функцію, щоб точно знати, на скільки позицій можна безпечно зсунути зразок після невдачі.

6. 1) Не зміниться, залишиться O(n*m). Алгоритм все одно має пройти по всіх можливих зсувах.

7. 1) Розбіжність буде виявлятися на першому ж символі для кожного зсуву, тому буде виконано лише `n-m+1` порівнянь.

8. 2) Кількість додаткової пам’яті не залежить від довжини тексту та зразка.

9. 2) Для кожного з `n` зсувів буде виконано одне порівняння.

10. 2) Зсув на 1 позицію після невдачі є неефективним, бо алгоритм не враховує, що `m-1` символів вже збіглися. Більш розумні алгоритми використовують цю інформацію.

Завдання для самостійної роботи

1. Реалізуйте версію наївного алгоритму, яка повертає масив з індексами всіх входжень зразка.

2. Модифікуйте алгоритм, щоб він виконував пошук без урахування регістру символів.

3. Напишіть функцію, яка підраховує точну кількість виконаних порівнянь символів.

4. Адаптуйте алгоритм для пошуку не рядка, а масиву цілих чисел всередині іншого масиву.

5. Реалізуйте пошук із “wildcard” символом (напр., `*`), який відповідає будь-якому символу.

6. Порівняйте на практиці продуктивність наївного пошуку на природному тексті (напр., уривок з книги) та на генетичній послідовності (рядок з ‘A’, ‘C’, ‘G’, ‘T’).

7. Напишіть програму, яка візуалізує процес “ковзного вікна”, показуючи зразок під текстом на кожній ітерації.

8. Оцініть, як вплине на продуктивність попереднє перетворення рядків у масиви чисел (кодів символів).

9. Реалізуйте наївний пошук справа наліво. Чи дасть це якусь перевагу?

10. Напишіть функцію, яка генерує текстові рядки, що створюють найгірший випадок для заданого зразка.

Теми лабораторних робіт

1. Порівняльний аналіз продуктивності наївного пошуку та алгоритму Кнута-Морріса-Пратта.

2. Розробка простого текстового редактора з функцією пошуку на основі наївного алгоритму.

3. Аналіз ефективності наївного пошуку на даних з різним розміром алфавіту.

4. Реалізація та тестування наївного пошуку для двовимірних масивів (пошук зображення-зразка у більшому зображенні).

5. Створення бенчмарку для демонстрації найгіршого випадку $O(m \cdot n)$.

6. Застосування наївного алгоритму для вирішення задач з біоінформатики (напр., пошук коротких генних послідовностей).

7. Оптимізація наївного пошуку для сучасних процесорних архітектур (використання SIMD-інструкцій).

8. Візуалізатор для порівняння роботи наївного алгоритму та алгоритму Боєра-Мура.

9. Розробка утиліти командного рядка (аналог `grep`) з використанням наївного пошуку.

10. Аналіз впливу кеш-пам’яті на продуктивність наївного алгоритму.

Тепер переходимо до біль цікавих алгоритмів пошуку підрядків. Ці алгоритми складні, тому будемо наводити покрокове пояснення (майже так само, як робили для алгоритмів роботи із деревами).

10.2.2. Алгоритм пошуку підрядка Кнута-Морріса-Пратта (КМП)

Теоретичні відомості

Задача пошуку підрядка є однією з фундаментальних задач в обробці тексту та інформатиці. Вона полягає в знаходженні всіх входжень певного шаблону (pattern) P довжиною m у тексті T довжиною n. Наївний алгоритм розв'язання цієї задачі має часову складність O(n·m), що є неефективним для великих текстів.

Алгоритм Кнута-Морріса-Пратта (КМП), розроблений у 1977 році, вирішує цю проблему, досягаючи лінійної часової складності O(n + m). Ключова ідея алгоритму полягає у використанні інформації про структуру самого шаблону для уникнення непотрібних порівнянь.

Префікс-функція: основа алгоритму КМП

Префікс-функція є центральним поняттям алгоритму КМП. Для розуміння її роботи спочатку визначимо необхідні терміни.

Префікс рядка - це підрядок, який починається з першого символу рядка. Наприклад, для рядка "ABCAB" префіксами є: "","A", "AB", "ABC", "ABCA", "ABCAB".

Суфікс рядка - це підрядок, який закінчується останнім символом рядка. Для того ж рядка "ABCAB" суфіксами є: "","B", "AB", "CAB", "BCAB", "ABCAB".

Власний префікс та власний суфікс - це префікс або суфікс, який не дорівнює всьому рядку.

Префікс-функція π(i) для позиції i в рядку P визначається як довжина найдовшого власного префікса підрядка P[0..i], який одночасно є його власним суфіксом.

Іншими словами, π(i) показує максимальну довжину k таку, що: - k < i + 1 (власний префікс) - P[0..k-1] = P[i-k+1..i] (префікс дорівнює суфіксу)

Розглянемо детальний приклад для шаблону P = "ABABCABAB":

```
Індекс i:     0  1  2  3  4  5  6  7  8
Символ P[i]:  A  B  A  B  C  A  B  A  B
π(i):         0  0  1  2  0  1  2  3  4
```

Пояснимо крок за кроком:

- i = 0: P[0] = "A". Немає власних префіксів, тому π(0) = 0.

- i = 1: P[0..1] = “AB”. Власні префікси: “A”. Власні суфікси: “B”. Немає спільних, тому π(1) = 0.

- i = 2: P[0..2] = “ABA”. Власні префікси: “A”, “AB”. Власні суфікси: “A”, “BA”. Спільний: “A” довжиною 1, тому π(2) = 1.

- i = 3: P[0..3] = “ABAB”. Спільний префікс-суфікс: “AB” довжиною 2, тому π(3) = 2.

- i = 4: P[0..4] = “ABABC”. Немає спільних префіксів-суфіксів, тому π(4) = 0.

- i = 5: P[0..5] = “ABABCA”. Спільний: “A” довжиною 1, тому π(5) = 1.

- i = 6: P[0..6] = “ABABCAB”. Спільний: “AB” довжиною 2, тому π(6) = 2.

- i = 7: P[0..7] = “ABABCABA”. Спільний: “ABA” довжиною 3, тому π(7) = 3.

- i = 8: P[0..8] = “ABABCABAB”. Спільний: “ABAB” довжиною 4, тому π(8) = 4.

Алгоритм побудови префікс-функції

Розглянемо еквівалентне поняття: $\pi[i]$ — довжина найдовшого **бордера** префікса $P[0..i]$. Бордер — це рядок, що є і префіксом, і суфіксом водночас.

Приклад короткої інтуїції:

- Якщо $\pi[i] = k$, то $P[0..k-1] = P[i-k+1..i]$. Тобто префікс довжини k «наклався» на кінець $P[0..i]$.

Під час пошуку π підказує, наскільки «відкотитися» в шаблоні, коли трапляється невідповідність символів.

Але щоб π була корисною, її потрібно вміти будувати за $O(m)$. Це можливо завдяки ключовій властивості бордерів:

- Якщо B — бордер деякого рядка S, то бордер будь-якого бордера B теж є бордером S.

Отже, якщо ми намагалися подовжити бордер довжини k, але новий символ не підійшов, варто спробувати коротший бордер довжини $\pi[k-1]$, потім $\pi[\pi[k-1]-1]$ і так далі. Ми рухаємось по «ланцюжку» бордерів, не повертаючись до невигідних варіантів.

3. Алгоритм побудови π: ідея та інваріант

Ми підтримуємо змінну k — поточну довжину «найкращого» бордера для вже обробленого префікса.

- На кожному кроці i намагаємось подовжити бордер: якщо $P[i] = P[k]$, збільшуємо k і кладемо $\pi[i] = k$.

- Якщо $P[i] \neq P[k]$, «скочуємось» за ланцюжком π: $k \leftarrow \pi[k-1]$, і знову пробуємо порівняння з $P[k]$. Якщо k дорівнює 0 і збіг неможливий, записуємо $\pi[i] = 0$.

-Інваріант циклу за i:

- Перед спробою порівняння $P[i]$ з $P[k]$ значення k дорівнює $\pi[i-1]$, тобто є довжиною найдовшого бордера для $P[0..i-1]$.

- Усі відступи $k \leftarrow \pi[k-1]$ зберігають інваріант, бо ми лишаємося всередині ланцюжка бордерів.

4. Псевдокод (нульова індексація, $m = |P|$)

```
function compute_pi(P)
    m ← length(P)
    π ← array of size m filled with 0
    k ← 0
    for i from 1 to m - 1
        while k > 0 and P[k] ≠ P[i]
            k ← π[k - 1]
        if P[k] = P[i]
            k ← k + 1
        π[i] ← k
    return π
```

Пояснення до рядків псевдокоду:

- `k` — поточна довжина збігу префікса з суфіксом (кандидат у бордер).

- Якщо символи не збігаються, «стискаємо» кандидата до його власного бордера `π[k-1]`.

- Якщо символи збігаються, подовжуємо збіг на один.

- Кожен символ рядка P «рухає» `k` вперед щонайбільше один раз і відступає назад по ланцюжку π теж щонайбільше один раз, тому сумарно $O(m)$.

Розглянем покроковий приклад 1: $P =$ “ababaca”:

- Коли ми читаємо символ `P[i]`, ми вже знаємо найдовший бордер для `P[0..i-1]` (це `k`).

- Якщо `P[i]` дорівнює `P[k]`, бордер подовжується.

- Якщо ні, пробуємо коротший бордер довжини `pi[k-1]`.

Хід обчислення такий:

- Початок

- pi = [0, 0, 0, 0, 0, 0, 0], k = 0

- i = 1, порівнюємо P[1] = ‘b’ з P[0] = ‘a’
- не збіг → k = 0, pi[1] = 0
- i = 2, порівнюємо P[2] = ‘a’ з P[0] = ‘a’
- збіг → k = 1, pi[2] = 1
- i = 3, порівнюємо P[3] = ‘b’ з P[1] = ‘b’
- збіг → k = 2, pi[3] = 2
- i = 4, порівнюємо P[4] = ‘a’ з P[2] = ‘a’
- збіг → k = 3, pi[4] = 3
- i = 5, порівнюємо P[5] = ‘c’ з P[3] = ‘b’
- не збіг → k = pi[2] = 1
- порівнюємо P[5] = ‘c’ з P[1] = ‘b’
- не збіг → k = pi[0] = 0
- k = 0 і збігу нема → pi[5] = 0
- i = 6, порівнюємо P[6] = ‘a’ з P[0] = ‘a’
- збіг → k = 1, pi[6] = 1

Підсумок:

- $\pi = [0,0,1,2,3,0,1]$

Розглянемо ще один покроковий приклад: $P =$ “aabaaab”:

- Початок
- pi = [0, 0, 0, 0, 0, 0, 0], k = 0
- i = 1, P[1] = ‘a’ проти P[0] = ‘a’
- збіг → k = 1, pi[1] = 1
- i = 2, P[2] = ‘b’ проти P[1] = ‘a’
- не збіг → k = pi[0] = 0
- k = 0, ‘b’ проти ‘a’ → збігу нема → pi[2] = 0
- i = 3, P[3] = ‘a’ проти P[0] = ‘a’
- збіг → k = 1, pi[3] = 1
- i = 4, P[4] = ‘a’ проти P[1] = ‘a’
- збіг → k = 2, pi[4] = 2
- i = 5, P[5] = ‘a’ проти P[2] = ‘b’

- не збіг → k = pi[1] = 1

- тепер P[5] = ‘a’ проти P[1] = ‘a’

- збіг → k = 2, pi[5] = 2

- i = 6, P[6] = ‘b’ проти P[2] = ‘b’

- збіг → k = 3, pi[6] = 3

Підсумок:

- $\pi = [0,1,0,1,2,2,3]$

Тепер необхідно розглянути питання, чому алгоритм працює за $O(m)$ часу?

- Кожне збільшення k може статись щонайбільше $m - 1$ разів (бо k зростає від 0 до $m - 1$).

- Кожне зменшення k через `k = pi[k-1]` також сукупно відбувається не більше $m - 1$ разів, бо k при відкатах ніколи не збільшується і не повертається до вищих значень без нового збігу символів.

- Отже, сумарно внутрішній `while` робить $O(m)$ ітерацій на весь алгоритм, а зовнішній `for` — рівно $m - 1$ кроків, разом $O(m)$.

9. Кути й крайові випадки, про які варто пам’ятати

1. Порожній рядок P зазвичай не розглядають. Якщо потрібно, домовляються, що π — порожній масив.

2. Рядок із усіх однакових символів, наприклад “aaaaa”:

- $\pi = [0,1,2,3,4]$. Це корисно для перевірки вашої реалізації.

- Рядок із попарно різних символів, наприклад “abcde”:

- $\pi = [0,0,0,0,0]$.

3. Реєстр символів:

- Якщо потрібно нечутливе до регістру порівняння, нормалізуйте P (і текст у пошуку) до одного регістру перед обчисленням π.

4. Юнікод:

- Алгоритм працює над послідовністю «символів» так, як вони представлені у вашій мові програмування. Для Python важливо, що `len` рахує code points, а не «графеми», що складаються з кількох code points.

5. Корисні наслідки π-функції

- Мінімальний період рядка P можна знайти так:

- Нехай $p = m - \pi[m-1]$.

- Якщо $m \bmod p = 0$, то мінімальний період дорівнює p, інакше період дорівнює m.

- Виявлення повторюваної структури:

- Якщо $\pi[m-1] > 0$, кінець P "резонує" з початком, що використовується в задачах аналізу шаблонів.

6. Еквівалентні формулювання та поширені помилки

- Не плутайте $\pi[i]$ з кількістю входжень шаблона в тексті — π обчислюється лише з шаблона P.

- Не намагайтеся «повертати» k через $k \leftarrow k-1$ — це дасть $O(m^2)$. Коректний відступ лише через ланцюжок бордерів `k = pi[k-1]`.

- Порядок дій у циклі критичний:

- Спочатку «відкати» `while k > 0 and p[k] != p[i]`

- Потім перевірка збігу `if p[k] == p[i]`

- Наприкінці присвоєння `pi[i] = k`

Алгоритм пошуку КМП

Тепер перейдемо до вивчення алгоритму КМП. Він вирішує фундаментальну проблему наївного підходу до пошуку підрядків. У наївному алгоритмі, коли відбувається невідповідність символів, ми повертаємося назад і починаємо порівняння з наступної позиції тексту. Це призводить до повторного порівняння символів, які ми вже перевірили.

КМП використовує спостереження: якщо ми вже порівняли частину шаблону з текстом і знаємо структуру самого шаблону, то при невідповідності можемо визначити, з якої позиції шаблону продовжити порівняння, не повертаючись назад по тексту.

```
Алгоритм KMP_DetailedSearch(T, P):
    Вхід:
        T - текст довжиною n
        P - шаблон довжиною m
    Вихід:
        positions - список позицій входжень P у T

    1. n ← довжина(T)
    2. m ← довжина(P)

    // Спеціальні випадки
```

```
    3. якщо m = 0:
    4.     повернути порожній список
    5. якщо m > n:
    6.     повернути порожній список

    // Обчислюємо префікс-функцію для шаблону
    7. π ← ComputePrefixFunction(P)

    // Ініціалізація
    8. positions ← порожній список
    9. q ← 0  // індекс поточного символу в шаблоні P

    // Основний цикл по тексту
    10. для i від 0 до n-1:
    11.     // Поки є невідповідність і ми не на початку шаблону
    12.     поки q > 0 і P[q] ≠ T[i]:
    13.         // Використовуємо префікс-функцію для визначення
    14.         // наступної позиції порівняння в шаблоні
    15.         q ← π[q-1]
    16.
    17.     // Перевіряємо відповідність поточних символів
    18.     якщо P[q] = T[i]:
    19.         // Символи співпали, переходимо до наступного символу
шаблону
    20.         q ← q + 1
    21.
    22.     // Перевіряємо, чи знайшли повне входження
    23.     якщо q = m:
    24.         // Додаємо позицію початку входження
    25.         додати (i - m + 1) до positions
    26.         // Продовжуємо пошук наступних входжень
    27.         // використовуючи префікс-функцію
    28.         q ← π[q-1]

    29. повернути positions
```

Покрокове пояснення роботи алгоритму:

Крок 1-6: Ініціалізація та перевірка граничних випадків

Спочатку визначаємо довжини тексту та шаблону. Перевіряємо спеціальні випадки: - Якщо шаблон порожній, повертаємо порожній список - Якщо шаблон довший за текст, входжень бути не може

Крок 7: Обчислення префікс-функції

Це ключовий етап попередньої обробки. Префікс-функція дозволяє нам знати, куди “стрибнути” при невідповідності.

Крок 9: Змінна q

Змінна `q` відстежує, скільки символів шаблону вже співпало з текстом. Це індекс наступного символу шаблону, який потрібно порівняти.

Крок 10-28: Основний цикл

Проходимо по кожному символу тексту. Для кожної позиції `i`:

Крок 12-15: Обробка невідповідності

Якщо `P[q] ≠ T[i]`, це означає невідповідність. Замість того, щоб почати з початку шаблону, використовуємо префікс-функцію: - `q ← π[q-1]` означає: “який найдовший префікс шаблону є також суфіксом вже перевіреної частини?” - Це дозволяє “перестрибнути” на позицію, де можливе часткове співпадіння

Крок 18-20: Обробка відповідності

Якщо символи співпали, збільшуємо `q`, переходячи до наступного символу шаблону.

Крок 23-28: Виявлення повного входження

Коли `q = m`, це означає, що всі `m` символів шаблону співпали: - Зберігаємо позицію початку входження: `i - m + 1` - Використовуємо префікс-функцію для продовження пошуку: `q ← π[q-1]`

Розглянемо пошук шаблону **P = “ABABCABAB”** у тексті **T = “ABABDABABCABABEF”**.

Крок 1: Обчислюємо префікс-функцію

```
i:    0  1  2  3  4  5  6  7  8
P[i]: A  B  A  B  C  A  B  A  B
π[i]: 0  0  1  2  0  1  2  3  4
```

Крок 2: Виконуємо пошук

```
Текст T:  A B A B D A B A B C A B A B E F
Індекси:  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

Ініціалізація: q = 0, i = 0

i=0, T[0]='A', q=0:
  P[0]='A' = T[0] ✓
  q = 1
```

```
i=1, T[1]='B', q=1:
  P[1]='B' = T[1] ✓
  q = 2

i=2, T[2]='A', q=2:
  P[2]='A' = T[2] ✓
  q = 3

i=3, T[3]='B', q=3:
  P[3]='B' = T[3] ✓
  q = 4

i=4, T[4]='D', q=4:
  P[4]='C' ≠ T[4] ✗
  Цикл while: q > 0 і P[4] ≠ 'D'
    q = π[3] = 2
  P[2]='A' ≠ T[4] ✗
  Цикл while: q > 0 і P[2] ≠ 'D'
    q = π[1] = 0
  P[0]='A' ≠ T[4] ✗
  q залишається 0

i=5, T[5]='A', q=0:
  P[0]='A' = T[5] ✓
  q = 1

i=6, T[6]='B', q=1:
  P[1]='B' = T[6] ✓
  q = 2

i=7, T[7]='A', q=2:
  P[2]='A' = T[7] ✓
  q = 3

i=8, T[8]='B', q=3:
  P[3]='B' = T[8] ✓
  q = 4

i=9, T[9]='C', q=4:
  P[4]='C' = T[9] ✓
  q = 5

i=10, T[10]='A', q=5:
  P[5]='A' = T[10] ✓
  q = 6
```

```
i=11, T[11]='B', q=6:
  P[6]='B' = T[11] ✓
  q = 7

i=12, T[12]='A', q=7:
  P[7]='A' = T[12] ✓
  q = 8

i=13, T[13]='B', q=8:
  P[8]='B' = T[13] ✓
  q = 9

  q = m = 9 → Знайдено входження!
  Позиція: i - m + 1 = 13 - 9 + 1 = 5
  Додаємо 5 до результату
  q = π[8] = 4

i=14, T[14]='E', q=4:
  P[4]='C' ≠ T[14] ✗
  Цикл while: q = π[3] = 2
  P[2]='A' ≠ T[14] ✗
  Цикл while: q = π[1] = 0
  P[0]='A' ≠ T[14] ✗
  q = 0

i=15, T[15]='F', q=0:
  P[0]='A' ≠ T[15] ✗
  q = 0

Результат: входження знайдено на позиції 5
```

Тепер розглянемо деякі ключові моменти.

1. Чому використовується π[q-1], а не π[q]?

Коли відбувається невідповідність на позиції **q**, це означає, що символи **P[0..q-1]** співпали з відповідною частиною тексту, але **P[q]** не співпав. Тому ми використовуємо **π[q-1]**, щоб знайти найдовший префікс з уже перевірених **q** символів.

2. Чому цикл while, а не if?

Цикл **while** потрібен, тому що після одного "стрибка" через префікс-функцію може знову виникнути невідповідність. Ми продовжуємо

“стрибати”, поки не знайдемо відповідність або не дійдемо до початку шаблону.

3. Обробка після знаходження входження

Після знаходження повного входження ми не починаємо з `q = 0`, а використовуємо `q = π[q-1]`. Це дозволяє ефективно знаходити входження, що перекриваються.

Аналіз складності

Часова складність

Основний цикл виконується `n` разів. Хоча всередині є цикл `while`, загальна кількість операцій зменшення `q` не може перевищувати загальну кількість його збільшень, яка обмежена `n`. Тому загальна складність O(n).

З урахуванням попередньої обробки O(m), повна складність: O(n + m).

Просторова складність

Потрібна додаткова пам’ять тільки для префікс-функції: O(m).

Варіації та оптимізації

1. Пошук першого входження

Якщо потрібно знайти тільки перше входження, можна модифікувати алгоритм:

```
якщо q = m:
    повернути (i - m + 1)  // повертаємо одразу
```

2. Підрахунок кількості входжень

Замість збереження позицій можна просто рахувати:

```
count ← 0
...
якщо q = m:
    count ← count + 1
    q ← π[q-1]
```

3. Пошук з обмеженням на перекриття

Якщо не потрібні входження, що перекриваються:

```
якщо q = m:
    додати (i - m + 1) до positions
    q ← 0  // починаємо з початку
```

Алгоритм КМП - це елегантне рішення класичної задачі, яке демонструє силу попередньої обробки та використання властивостей самої структури даних для оптимізації пошуку.

Область застосування

Алгоритм КМП широко використовується в різних галузях:

- Текстові редактори та процесори - для функцій пошуку та заміни тексту

- Біоінформатика - для пошуку патернів у ДНК та білкових послідовностях

- Системи виявлення вторгнень - для пошуку сигнатур атак у мережевому трафіку

- Компілятори - для лексичного аналізу та розпізнавання токенів

- Пошукові системи - як базовий компонент для більш складних алгоритмів

- Обробка логів - для виявлення певних патернів у файлах журналів

- Антивірусне програмне забезпечення - для пошуку сигнатур вірусів

Приклади застосування

Приклад 1: Пошук у ДНК послідовності

Припустимо, ми шукаємо ген з послідовністю “ATCGATCG” у фрагменті ДНК:

```
ДНК: GATCGATCGATCGATCGATCG
Ген: ATCGATCG
```

Використовуючи КМП, ми знаходимо входження на позиціях 1, 5, 9 та 13.

Приклад 2: Виявлення спаму

Система фільтрації спаму може використовувати КМП для пошуку підозрілих фраз:

```
Текст листа: "Вітаємо! Ви виграли мільйон! Натисніть тут, щоб отримати виграш!"
Спам-фраза: "виграли мільйон"
```

Приклад 3: Валідація формату даних

Перевірка, чи містить рядок певний формат дати:

```
Текст: "Зустріч призначена на 2024-03-15 о 14:30"
Шаблон: "20[0-9][0-9]-[0-1][0-9]-[0-3][0-9]"
```

Переваги та недоліки

Переваги

- Лінійна часова складність - O(n + m), що є оптимальним для задачі пошуку підрядка

- Відсутність зворотних переходів - алгоритм ніколи не повертається назад по тексту

- Попередня обробка тільки шаблону - не потребує попередньої обробки тексту

- Ефективність для повторюваних шаблонів - особливо ефективний для шаблонів з багатьма повтореннями

- Простота реалізації - відносно простий для розуміння та імплементації

- Гарантована продуктивність - немає найгіршого випадку з квадратичною складністю

Недоліки

- Додаткова пам’ять - потребує O(m) додаткової пам’яті для зберігання префікс-функції

- Попередня обробка - вимагає часу O(m) на обчислення префікс-функції

- Неефективність для коротких шаблонів - для дуже коротких шаблонів наївний алгоритм може бути швидшим через меншу кількість операцій

- Складність для розуміння - концепція префікс-функції може бути складною для початківців

- Неоптимальність для випадкових текстів - алгоритм Бойєра-Мура може бути швидшим для випадкових текстів

Контрольні запитання

1. Що таке префікс-функція і яке її призначення в алгоритмі КМП?

2. Яка часова складність алгоритму КМП і чому він ефективніший за наївний підхід?

3. Поясніть, чому алгоритм КМП ніколи не повертається назад по тексту?

4. Для рядка “AABAABA” обчисліть префікс-функцію вручну.

5. В яких випадках алгоритм КМП може бути менш ефективним порівняно з іншими алгоритмами пошуку?

6. Яка роль циклу while у алгоритмах побудови префікс-функції та пошуку КМП?

7. Чому префікс-функція для позиції 0 завжди дорівнює 0?

8. Як алгоритм КМП обробляє випадок, коли знайдено входження шаблону?

9. Яка просторова складність алгоритму КМП?

10. Наведіть приклад шаблону, для якого префікс-функція матиме максимальні значення.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму пошуку підрядка Кнута-Морріса-Пратта

2. Мета

Вивчити принципи роботи алгоритму КМП, навчитися обчислювати префікс-функцію та реалізувати ефективний пошук підрядків у тексті. Порівняти продуктивність з наївним алгоритмом пошуку.

3. Завдання

- Реалізувати функцію обчислення префікс-функції

- Реалізувати алгоритм пошуку КМП

- Реалізувати наївний алгоритм пошуку для порівняння

- Створити тестові випадки для перевірки коректності

- Провести порівняльний аналіз продуктивності алгоритмів

- Візуалізувати процес пошуку покроково

4. Опис

Студенти повинні реалізувати повноцінну систему пошуку підрядків з використанням алгоритму КМП. Програма повинна включати: - Модуль обчислення префікс-функції з виведенням проміжних результатів - Модуль пошуку з можливістю знаходження всіх входжень - Інтерфейс для введення тексту та шаблону - Візуалізацію кроків алгоритму - Статистику порівнянь символів

5. Порядок виконання

- Створити клас або модуль StringMatcher

- Реалізувати метод computePrefixFunction згідно з наведеним псевдокодом

- Реалізувати метод KMPSearch для пошуку всіх входжень

- Додати метод naiveSearch для порівняння

- Створити метод compareAlgorithms для аналізу продуктивності

- Реалізувати візуалізацію процесу пошуку

- Протестувати на різних видах даних

6. Результат

Програма повинна: - Знаходити всі входження шаблону в тексті - Виводити префікс-функцію для заданого шаблону - Показувати кількість порівнянь символів - Візуалізувати процес пошуку покроково - Порівнювати час виконання з наївним алгоритмом

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функція обчислення префікс-функції computePrefixFunction

- Функція пошуку КМП KMPSearch

- Функція наївного пошуку naiveSearch

- Клас або структура для зберігання результатів пошуку

- Функції візуалізації та статистики

2. Продемонструвати роботу додатку:

- Пошук у короткому тексті з візуалізацією кроків

- Пошук множинних входжень шаблону

- Порівняння продуктивності на великих текстах

- Обробку граничних випадків (порожній шаблон, шаблон довший за текст)

- Виведення статистики порівнянь

3. Бути готовим відповісти на питання:

- Як обчислюється префікс-функція для конкретного прикладу?

- Чому алгоритм має лінійну складність?

- В яких випадках КМП працює найефективніше?

- Як модифікувати алгоритм для пошуку з урахуванням регістру?

- Як адаптувати алгоритм для пошуку кількох шаблонів одночасно?

Тести

1. Префікс-функція для рядка “AAAA” має вигляд:

1) [0, 0, 0, 0]

2) [0, 1, 2, 3]

3) [1, 2, 3, 4]

4) [0, 1, 1, 1]

2. Часова складність алгоритму КМП становить:

1) $O(n^2)$

2) $O(n \cdot m)$

3) $O(n + m)$

4) $O(n \log m)$

3. Основна перевага КМП над наївним алгоритмом полягає в:

1) Меншому використанні пам’яті

2) Простішій реалізації

3) Відсутності повернення назад по тексту

4) Кращій роботі з короткими шаблонами

4. Для шаблону “ABCABC” значення $\pi[5]$ дорівнює:

1) 0

2) 1

3) 2

4) 3

5. В алгоритмі КМП префікс-функція обчислюється для:

1) Тексту

2) Шаблону

3) Об’єднання тексту і шаблону

4) Кожного символу окремо

Відповіді:

1. b)

2. c)

3. c)

4. c)

5. b)

Тести підвищеної складності

1. Яке максимальне значення може мати префікс-функція $\pi[i]$ для позиції i?

1) i

2) i + 1

3) i - 1

4) m - 1

2. Для якого типу шаблонів алгоритм КМП працює найгірше з точки зору кількості порівнянь?

1) Шаблони з багатьма повтореннями

2) Випадкові шаблони без повторень

3) Шаблони виду “AAA…A”

4) Шаблони виду “ABCD…XYZ”

3. Скільки разів виконається внутрішній цикл while при обчисленні префікс-функції для рядка довжиною m в найгіршому випадку?

1) O(m)

2) $O(m^2)$

3) O(m log m)

4) O(2m)

4. Який результат дасть префікс-функція для рядка “ABACABAB”?

1) [0, 0, 1, 0, 1, 2, 3, 2]

2) [0, 0, 1, 0, 1, 2, 1, 2]

3) [0, 0, 1, 0, 1, 2, 3, 4]

4) [0, 0, 0, 0, 1, 2, 3, 2]

5. Якщо при пошуку відбулася невідповідність на позиції q шаблону, то наступне порівняння відбудеться з позиції:

1) 0

2) q - 1

3) π[q]

4) π[q - 1]

6. Для шаблону виду “A^k B A^k” (k символів A, потім B, потім знову k символів A), чому дорівнює π[2k]?

1) k

2) k - 1

3) k + 1

4) 2k

7. Яка амортизована складність одного кроку в основному циклі алгоритму КМП?

1) O(1)

2) O(log m)

3) O(m)

4) O(n)

8. В якому випадку алгоритм КМП виконає найменшу кількість порівнянь символів?

1) Коли перший символ шаблону не зустрічається в тексті

2) Коли шаблон складається з одного символу

3) Коли текст і шаблон однакові

4) Коли шаблон довший за текст

9. Як зміниться префікс-функція, якщо до кінця рядка додати символ, якого в ньому не було?

1) Всі значення залишаться незмінними, останнє буде 0

2) Всі значення збільшаться на 1

3) Деякі значення можуть змінитися

4) Тільки останнє значення зміниться

10. Для реалізації пошуку кількох шаблонів одночасно краще використовувати:

1) КМП для кожного шаблону окремо

2) Модифікований КМП з об’єднаними шаблонами

3) Алгоритм Ахо-Корасік

4) Наївний алгоритм з оптимізаціями

Відповіді:

1. a) i - Максимальне значення $\pi[i] = i$, коли весь префікс довжини i є періодичним

2. b) - Випадкові шаблони без повторень мають малі значення префікс-функції, що зменшує ефективність

3. a) O(m) - Амортизована складність лінійна завдяки властивостям префікс-функції

4. a) [0, 0, 1, 0, 1, 2, 3, 2] - Обчислюється покроково згідно з визначенням префікс-функції

5. d) π[q - 1] - При невідповідності на позиції q переходимо до π[q-1]

6. a) k - Префікс “A^k” довжини k співпадає з суфіксом “A^k”

7. a) O(1) - Кожен символ тексту обробляється константну кількість разів

8. a) - Якщо перший символ не зустрічається, виконується тільки n порівнянь

9. a) - Новий унікальний символ не може утворити префікс-суфікс

10. c) - Алгоритм Ахо-Корасік спеціально розроблений для пошуку багатьох шаблонів

Завдання для самостійної роботи

1. Реалізуйте модифікацію алгоритму КМП для пошуку з урахуванням регістру символів та можливістю ігнорування регістру за вибором користувача.

2. Створіть версію алгоритму КМП, яка знаходить не тільки точні входження, але й входження з однією помилкою (заміна, вставка або видалення одного символу).

3. Розробіть візуалізатор алгоритму КМП, який покроково показує процес обчислення префікс-функції та пошуку з підсвічуванням поточних позицій.

4. Імплементуйте алгоритм для пошуку найдовшого спільного підрядка двох рядків, використовуючи ідеї з алгоритму КМП.

5. Створіть програму для аналізу періодичності рядків, використовуючи властивості префікс-функції для виявлення повторюваних патернів.

6. Реалізуйте потоковий варіант алгоритму КМП, який може обробляти текст, що надходить частинами, без збереження всього тексту в пам’яті.

7. Розробіть модифікацію алгоритму для пошуку шаблонів з символами підстановки (wildcard), де ‘?’ означає будь-який один символ.

8. Створіть бенчмарк для порівняння КМП з алгоритмами Бойєра-Мура та Рабіна-Карпа на різних типах даних.

9. Імплементуйте двовимірний варіант алгоритму КМП для пошуку двовимірних патернів у матрицях символів.

10. Розробіть систему виявлення плагіату в текстах, використовуючи алгоритм КМП для пошуку спільних фрагментів між документами.

Теми лабораторних робіт

1. Розробка текстового редактора з функцією пошуку та заміни на основі алгоритму КМП з підтримкою регулярних виразів.

2. Створення системи моніторингу логів з використанням КМП для виявлення критичних помилок та аномалій у реальному часі.

3. Імплементація біоінформатичного інструменту для пошуку генетичних послідовностей у геномах з візуалізацією результатів.

4. Розробка антиплагіатної системи для перевірки студентських робіт з використанням модифікованого алгоритму КМП.

5. Створення музичного аналізатора для виявлення повторюваних мелодійних патернів у MIDI-файлах.

6. Реалізація системи фільтрації спаму з навчанням на основі виявлення характерних текстових патернів.

7. Розробка інструменту для аналізу та оптимізації SQL-запитів шляхом виявлення повторюваних підзапитів.

8. Створення гри “Пошук слів” з автоматичною генерацією головоломок та перевіркою рішень через КМП.

9. Імплементація системи виявлення вторгнень (IDS) для аналізу мережевого трафіку на основі сигнатур атак.

10. Розробка компресора тексту, який використовує префікс-функцію для виявлення та кодування повторюваних фрагментів.

10.1.3. Алгоритм пошуку підрядка Рабіна-Карпа

Теоретичні відомості

Алгоритм Рабіна-Карпа, розроблений Майклом Рабіном та Річардом Карпом у 1987 році, є ефективним алгоритмом пошуку підрядка, який використовує хешування для швидкого порівняння рядків. Основна ідея полягає в тому, щоб замість посимвольного порівняння рядків порівнювати їх хеш-значення, що може бути виконано за константний час.

Алгоритм особливо ефективний при пошуку множинних шаблонів або при роботі з великими алфавітами. На відміну від алгоритму КМП, який використовує детермінований підхід з префікс-функцією, Рабін-Карп застосовує ймовірнісний підхід через хешування.

Хеш-функція - це функція, яка перетворює рядок довільної довжини на число фіксованого розміру. Для алгоритму Рабіна-Карпа використовується поліноміальна хеш-функція, яка обчислюється за формулою:

`hash(s[0..m-1]) = (s[0] × d^(m-1) + s[1] × d^(m-2) + ... + s[m-1] × d^0) mod q`

Де: - s[0..m-1] - рядок довжиною m - d - розмір алфавіту (наприклад, 256 для ASCII) - q - велике просте число для зменшення діапазону хеш-значень - mod - операція знаходження остачі від ділення

Ковзне хешування

Ключова оптимізація алгоритму - це використання ковзного хешування. Замість повного перерахунку хешу для кожного підрядка тексту, ми можемо обчислити новий хеш на основі попереднього за константний час.

Якщо ми знаємо хеш для підрядка T[i..i+m-1], то хеш для T[i+1..i+m] обчислюється за формулою:

`hash_new = (d × (hash_old - T[i] × d^(m-1)) + T[i+m]) mod q`

Ця формула “видаляє” вплив першого символу старого вікна і “додає” вплив нового символу.

Алгоритм методу такий:

```
Алгоритм RabinKarp(T, P, d, q):
    Вхід:
        T - текст довжиною n
        P - шаблон довжиною m
        d - розмір алфавіту
        q - просте число для модульної арифметики
    Вихід:
        positions - список позицій входжень P у T

    1. n ← довжина(T)
    2. m ← довжина(P)
    3. positions ← порожній список

    // Перевірка граничних випадків
    4. якщо m = 0 або m > n:
    5.     повернути positions

    // Обчислення d^(m-1) mod q для ковзного хешування
    6. h ← 1
    7. для i від 1 до m-1:
    8.     h ← (h × d) mod q
```

```
// Обчислення початкових хешів
9. p ← 0  // хеш шаблону
10. t ← 0  // хеш першого вікна тексту

11. для i від 0 до m-1:
12.     p ← (d × p + P[i]) mod q
13.     t ← (d × t + T[i]) mod q

// Пошук входжень
14. для s від 0 до n-m:
15.     // Перевірка збігу хешів
16.     якщо p = t:
17.         // Перевірка фактичного збігу символів
18.         match ← true
19.         для j від 0 до m-1:
20.             якщо P[j] ≠ T[s+j]:
21.                 match ← false
22.                 вийти з циклу
23.
24.         якщо match = true:
25.             додати s до positions
26.
27.     // Обчислення хешу наступного вікна
28.     якщо s < n-m:
29.         t ← (d × (t - T[s] × h) + T[s+m]) mod q
30.
31.         // Корекція від'ємних значень
32.         якщо t < 0:
33.             t ← t + q

34. повернути positions
```

Наведемо пояснення кроків алгоритму:

Кроки 6-8: Обчислення $h = d^{(m-1)} \, mod \, q$

Це значення використовується для ефективного оновлення хешу. Ми обчислюємо його один раз, щоб уникнути повторних обчислень степеня.

Кроки 11-13: Обчислення початкових хешів

Обчислюємо хеш шаблону та хеш першого вікна тексту розміром m. Використовуємо схему Горнера для ефективного обчислення поліноміального хешу.

Кроки 16-25: Перевірка збігу

Коли хеші співпадають, це може бути або справжнє входження, або хеш-колізія (різні рядки з однаковим хешем). Тому необхідна додаткова перевірка символів.

Кроки 29-33: Ковзне хешування

Формула оновлення хешу видаляє вплив першого символу старого вікна і додає вплив нового символу. Корекція від'ємних значень необхідна через модульну арифметику.

Розглянемо пошук шаблону P = "ABC" у тексті T = "ABABCABC" з параметрами d = 256, q = 101.

```
Крок 1: Ініціалізація
n = 8, m = 3
h = d^(m-1) mod q = 256^2 mod 101 = 65536 mod 101 = 79

Крок 2: Обчислення хешу шаблону "ABC"
p = 0
p = (256 × 0 + 65) mod 101 = 65
p = (256 × 65 + 66) mod 101 = 16706 mod 101 = 47
p = (256 × 47 + 67) mod 101 = 12099 mod 101 = 85

Крок 3: Обчислення хешу першого вікна "ABA"
t = 0
t = (256 × 0 + 65) mod 101 = 65
t = (256 × 65 + 66) mod 101 = 47
t = (256 × 47 + 65) mod 101 = 12097 mod 101 = 83

Крок 4: Пошук
s = 0: t = 83, p = 85 → немає збігу
       Оновлення: t = (256 × (83 - 65 × 79) + 66) mod 101
                  t = (256 × (83 - 5135) + 66) mod 101
                  t = (256 × (-5052) + 66) mod 101
                  t = 48 (після корекції)

s = 1: t = 48, p = 85 → немає збігу
       Оновлення: t = (256 × (48 - 66 × 79) + 67) mod 101
                  t = 85

s = 2: t = 85, p = 85 → збіг хешів!
       Перевірка: "ABC" = "ABC" ✓
       Знайдено входження на позиції 2

s = 3: продовжуємо аналогічно...
```

Просте число q вибирається таким чином, щоб: - q × d поміщалося в машинне слово (для ефективності) - q було достатньо великим для зменшення ймовірності колізій - Типові значення: $2^{31} - 1, 10^9 + 7$

Основа d зазвичай дорівнює розміру алфавіту: - Для ASCII: d = 256 - Для малих літер англійського алфавіту: d = 26 - Для цифр: d = 10

Розглянемо деякі модифікації алгоритму.

1. Подвійне хешування

Для зменшення ймовірності колізій можна використовувати дві різні хеш-функції:

```
Алгоритм RabinKarpDoubleHash(T, P, d1, q1, d2, q2):
    // Обчислюємо два хеші для шаблону та вікон тексту
    // Входження підтверджується, якщо обидва хеші співпадають
```

2. Пошук множинних шаблонів

```
Алгоритм RabinKarpMultiple(T, patterns[], d, q):
    Вхід:
        T - текст
        patterns[] - масив шаблонів
        d, q - параметри хешування

    1. Створити хеш-таблицю patternHashes
    2. для кожного pattern в patterns:
    3.     hash ← обчислити хеш(pattern)
    4.     додати (hash, pattern) до patternHashes

    5. для кожного вікна в T:
    6.     windowHash ← обчислити хеш вікна
    7.     якщо windowHash є в patternHashes:
    8.         перевірити фактичний збіг
```

Цей алгоритм можна ефективно застосовувати для обробки великих текстів або потокової обробки:

```
Алгоритм StreamingRabinKarp(stream, P, d, q):
    1. Ініціалізувати циклічний буфер розміром m
    2. поки є дані в stream:
    3.     зчитати наступний символ
    4.     оновити буфер та хеш
    5.     перевірити на збіг з шаблоном
```

Область застосування

Алгоритм Рабіна-Карпа особливо ефективний у таких випадках:

- Пошук множинних шаблонів - коли потрібно знайти багато різних шаблонів у тексті

- Виявлення плагіату - порівняння документів на наявність ідентичних фрагментів

- Біоінформатика - пошук генетичних послідовностей з можливими мутаціями

- Детекція спаму - швидка перевірка наявності заборонених фраз

- Пошук у потокових даних - обробка даних, що надходять в реальному часі

- Пошук подібності - знаходження майже ідентичних фрагментів

- Криптографія - пошук патернів у зашифрованих даних

- Обробка зображень - пошук патернів у двовимірних масивах

Приклади застосування

Приклад 1: Детекція копіювання коду

Система перевірки студентських робіт шукає ідентичні фрагменти коду:

```
Код 1: "for(int i=0; i<n; i++) sum+=arr[i];"
Код 2: "int total=0; for(int i=0; i<n; i++) sum+=arr[i]; return total;"
Шаблон знайдено на позиції 11 в Коді 2
```

Приклад 2: Фільтрація URL

Перевірка URL на наявність заборонених доменів:

```
URL: https://example.com/malicious-site/download.exe
Заборонені: ["malicious-site", "virus", "trojan"]
Знайдено заборонений фрагмент "malicious-site"
```

Приклад 3: Пошук у ДНК з варіаціями

Пошук гена з можливими мутаціями використовує модифікований алгоритм з нечітким порівнянням.

Переваги та недоліки

Переваги

- Простота реалізації - алгоритм легко зрозуміти та реалізувати

- Ефективність для множинного пошуку - легко адаптується для пошуку багатьох шаблонів

- Низька просторова складність - O(1) додаткової пам’яті для базової версії

- Хороша середня продуктивність - O(n+m) в середньому випадку

- Природна паралелізація - різні частини тексту можна обробляти незалежно

- Адаптивність - легко модифікується для різних задач

Недоліки

- Гірший випадок O(nm) - при багатьох хеш-колізіях

- Залежність від вибору параметрів - неправильний вибір q може погіршити продуктивність

- Додаткові перевірки - необхідність перевірки символів при збігу хешів

- Можливість колізій - різні рядки можуть мати однаковий хеш

- Складність налагодження - помилки в хешуванні важко виявити

- Обмеження на розмір чисел - переповнення при великих m

Контрольні запитання

1. Що таке ковзне хешування і яка його роль в алгоритмі Рабіна-Карпа?

2. Чому необхідна додаткова перевірка символів після збігу хешів?

3. Як вибір простого числа q впливає на продуктивність алгоритму?

4. Яка часова складність алгоритму в найкращому та найгіршому випадках?

5. Як модифікувати алгоритм для пошуку всіх циклічних зсувів шаблону?

6. Чому використовується модульна арифметика в хеш-функції?

7. Як обчислюється хеш для наступного вікна тексту?

8. В яких випадках Рабін-Карп ефективніший за КМП?

9. Як зменшити ймовірність хеш-колізій?

10. Яка роль параметра d в алгоритмі?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму пошуку підрядка Рабіна-Карпа з різними хеш-функціями

2. Мета

Вивчити принципи хешування рядків, реалізувати алгоритм Рабіна-Карпа, дослідити вплив параметрів на продуктивність та порівняти з іншими алгоритмами пошуку.

3. Завдання

- Реалізувати базовий алгоритм Рабіна-Карпа
- Реалізувати версію з подвійним хешуванням
- Створити модуль для пошуку множинних шаблонів
- Дослідити вплив вибору q на кількість колізій
- Порівняти продуктивність з КМП та наївним алгоритмом
- Візуалізувати процес ковзного хешування

4. Опис

Студенти розробляють комплексну систему пошуку підрядків на основі алгоритму Рабіна-Карпа. Програма повинна демонструвати різні аспекти алгоритму: від базової реалізації до оптимізованих версій для спеціальних випадків. Особлива увага приділяється дослідженню хеш-колізій та методам їх мінімізації.

5. Порядок виконання

- Створити клас RabinKarpMatcher з базовою функціональністю
- Реалізувати метод computeHash для обчислення поліноміального хешу
- Реалізувати метод rollingHash для ефективного оновлення хешу
- Створити метод search для пошуку одного шаблону
- Розширити для пошуку множинних шаблонів
- Додати статистику колізій та порівнянь
- Реалізувати візуалізацію кроків алгоритму
- Провести експерименти з різними параметрами

6. Результат

Готова програма повинна: - Знаходити всі входження шаблону(ів) у тексті - Відображати кількість хеш-колізій - Показувати процес ковзного хешування - Порівнювати різні варіанти хеш-функцій - Демонструвати залежність продуктивності від параметрів - Обробляти тексти різних розмірів та алфавітів

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
- Функція обчислення поліноміального хешу computeHash
- Функція ковзного хешування rollingHash
- Функція пошуку search з обробкою колізій
- Модуль для множинного пошуку multiSearch
- Функції для різних хеш-функцій (поліноміальна, подвійна)
- Модуль статистики та візуалізації
2. Продемонструвати роботу додатку:
- Базовий пошук з виведенням хеш-значень
- Випадок з хеш-колізіями та їх обробкою
- Пошук множинних шаблонів одночасно
- Порівняння часу виконання для різних розмірів тексту
- Вплив параметрів d та q на продуктивність
- Візуалізацію ковзного хешування
3. Бути готовим відповісти на питання:
- Як обчислюється ковзний хеш для конкретного прикладу?
- Чому виникають колізії і як їх мінімізувати?
- Як вибрати оптимальні параметри для конкретної задачі?
- Як модифікувати алгоритм для нечіткого пошуку?
- Які переваги дає подвійне хешування?

Тести

1. Часова складність алгоритму Рабіна-Карпа в середньому випадку:
1) $O(n^2)$
2) $O(nm)$
3) $O(n+m)$
4) $O(n \log m)$
2. Основна перевага ковзного хешування полягає в:
1) Зменшенні кількості колізій
2) Обчисленні нового хешу за $O(1)$
3) Збільшенні точності порівняння

4) Зменшенні використання пам’яті

3. При яких умовах алгоритм Рабіна-Карпа має найгіршу складність O(nm)?

1) Коли всі символи тексту однакові

2) Коли шаблон довший за текст

3) При багатьох хеш-колізіях

4) При використанні великого простого числа

4. Формула ковзного хешування видаляє вплив:

1) Останнього символу старого вікна

2) Першого символу старого вікна

3) Середнього символу вікна

4) Всіх символів старого вікна

5. Чому в алгоритмі використовується модульна арифметика?

1) Для прискорення обчислень

2) Для забезпечення унікальності хешів

3) Для обмеження діапазону хеш-значень

4) Для спрощення реалізації

Відповіді:

1. c)

2. b)

3. c)

4. b)

5. c)

Тести підвищеної складності

1. Яка ймовірність хеш-колізії для двох випадкових рядків довжини m при використанні модуля q?

1) 1/q

2) m/q

3) 1/q^m

4) приблизно 1/q

2. Для тексту з n символів та k шаблонів довжини m, яка складність пошуку всіх шаблонів?

1) O(n + km)

2) O(nk + m)

3) O(nkm)

4) O(n + k + m)

3. При якому значенні d хеш-функція матиме найкращі властивості для тексту з алфавіту розміром σ?

1) d = 2

2) d = σ

3) $d = \sigma^2$

4) d = просте число більше σ

4. Скільки різних хеш-значень може мати рядок довжини m при фіксованих d та q?

1) d^m

2) q

3) min(d^m, q)

4) d^m mod q

5. Як зміниться алгоритм для пошуку шаблону з символом підстановки '?'?

1) Потрібно модифікувати хеш-функцію

2) Потрібно ігнорувати позиції з '?'

3) Алгоритм не підходить для цієї задачі

4) Потрібно використовувати множинні хеші

6. Для зменшення ймовірності колізій до $1/n^2$ потрібно вибрати q:

1) q > n

2) $q > n^2$

3) $q > m^2$

4) q > nm

7. При подвійному хешуванні з модулями q_1 та q_2 ймовірність колізії становить:

1) $1/\max(q_1, q_2)$

2) $1/(q_1 + q_2)$

3) приблизно $1/(q_1 \times q_2)$

4) $\min(1/q_1, 1/q_2)$

8. Яка модифікація алгоритму найефективніша для пошуку всіх циклічних зсувів шаблону?

1) Запустити алгоритм m разів

2) Подвоїти шаблон і шукати оригінал

3) Використати спеціальну хеш-функцію

4) Шукати в тексті T + T

9. При використанні 64-бітних чисел без модуля, для якої максимальної довжини шаблону гарантовано не буде переповнення при d=256?

1) 8

2) 7

3) 16

4) 4

10. Для потокової обробки даних найкраще підходить:

1) Класичний Рабін-Карп

2) Рабін-Карп з буферизацією

3) КМП

4) Наївний алгоритм

Відповіді:

1. d) приблизно 1/q - Для випадкових рядків хеші розподілені майже рівномірно

2. a) O(n + km) - O(km) для хешування шаблонів + O(n) для проходу по тексту

3. d) d = просте число більше σ - Прості числа дають кращий розподіл хешів

4. c) min(d^m, q) - Обмежено або кількістю можливих рядків, або модулем

5. c) - Рабін-Карп базується на точному порівнянні хешів, що неможливо з підстановками

6. b) $q > n^2$ - За принципом дня народження для n^2 пар рядків

7. c) приблизно $1/(q_1 \times q_2)$ - Колізія потрібна для обох хешів одночасно

8. b) - Шукаємо P в PP, всі входження крім першого - циклічні зсуви

9. b) 7 - 256^8 > 2^64, але 256^7 < 2^64

10. b) - Потрібен буфер розміру m для зберігання поточного вікна

Завдання для самостійної роботи

1. Реалізуйте версію алгоритму Рабіна-Карпа для двовимірного пошуку (пошук прямокутної матриці символів у більшій матриці).

2. Створіть адаптивний алгоритм, який автоматично вибирає оптимальні параметри d та q залежно від характеристик тексту та шаблону.

3. Розробіть модифікацію для приблизного пошуку, яка знаходить підрядки з хеш-значеннями, що відрізняються не більше ніж на задану величину.

4. Імплементуйте паралельну версію алгоритму для багатоядерних процесорів з розподілом тексту між потоками.

5. Створіть гібридний алгоритм, який використовує Рабіна-Карпа для довгих шаблонів та інший метод для коротких.

6. Реалізуйте версію з трьома різними хеш-функціями для мінімізації ймовірності колізій до negligible рівня.

7. Розробіть алгоритм для пошуку найдовшого спільного підрядка двох текстів, використовуючи ідеї Рабіна-Карпа.

8. Створіть модифікацію для роботи з Unicode-текстами з правильною обробкою багатобайтових символів.

9. Імплементуйте версію алгоритму для пошуку шаблонів змінної довжини (наприклад, слів між пробілами).

10. Розробіть систему індексації великих текстових корпусів на основі хешів Рабіна-Карпа для швидкого пошуку.

Теми лабораторних робіт

1. Розробка системи виявлення плагіату в програмному коді з використанням Рабіна-Карпа та синтаксичного аналізу.

2. Створення пошукової системи для геномних баз даних з підтримкою пошуку з мутаціями та індексацією.

3. Імплементація фільтра спаму реального часу для email-сервера з динамічним оновленням бази заборонених фраз.

4. Розробка інструменту для пошуку дублікатів у великих колекціях зображень через хешування блоків пікселів.

5. Створення системи моніторингу соціальних мереж для виявлення поширення певних повідомлень чи фраз.

6. Реалізація криптографічного аналізатора для пошуку патернів у зашифрованих повідомленнях.

7. Розробка музичного пошуковика, який знаходить мелодії за фрагментом через хешування нотних послідовностей.

8. Створення системи контролю версій з ефективним виявленням змін через блочне хешування файлів.

9. Імплементація веб-краулера з дедуплікацією контенту на основі хешування параграфів тексту.

10. Розробка інтелектуальної системи автодоповнення для IDE з використанням хешування кодових патернів.

10.2.4. Алгоритм пошуку підрядка Боєра-Мура

Теоретичні відомості

Алгоритм Боєра-Мура, розроблений Робертом Боєром та Дж. Струзером Муром у 1977 році, є одним з найефективніших алгоритмів пошуку підрядка в практичному застосуванні. Його революційна ідея полягає в тому, що порівняння символів шаблону з текстом відбувається справа наліво, а не зліва направо, як у більшості інших алгоритмів. Це дозволяє пропускати значні частини тексту без перевірки кожного символу.

Алгоритм особливо ефективний для довгих шаблонів та великих алфавітів, оскільки в середньому випадку його складність становить O(n/m), що краще за лінійну складність. Це досягається завдяки двом ключовим евристикам: евристиці поганого символу (bad character rule) та евристиці доброго суфікса (good suffix rule).

Основні концепції та евристики

Евристика поганого символу.

Евристика поганого символу базується на простій ідеї: якщо під час порівняння знайдено невідповідність, ми можемо зсунути шаблон так, щоб символ тексту, який спричинив невідповідність, вирівнявся з найправішим входженням цього символу в шаблоні. Якщо такого символу в шаблоні немає, шаблон можна зсунути повністю за позицію невідповідності.

Для ефективної реалізації цієї евристики створюється таблиця зсувів для кожного символу алфавіту. Ця таблиця показує, на скільки позицій можна зсунути шаблон при зустрічі кожного символу.

Евристика доброго суфікса.

Евристика доброго суфікса використовує інформацію про частину шаблону, яка вже співпала з текстом. Якщо суфікс шаблону співпав з текстом, але виникла невідповідність, ми шукаємо інше входження цього суфікса в шаблоні або найдовший префікс шаблону, який є суфіксом співпавшої частини.

Попередня обробка для евристики поганого символу

```
Алгоритм PreprocessBadCharacter(P, Σ):
    Вхід:
        P - шаблон довжиною m
        Σ - алфавіт
    Вихід:
        badChar - таблиця зсувів для кожного символу

    1. m ← довжина(P)
    2. для кожного символу c в Σ:
    3.     badChar[c] ← m  // За замовчуванням зсув дорівнює довжині
шаблону

    4. для i від 0 до m-2:  // Останній символ не обробляємо
    5.     badChar[P[i]] ← m - 1 - i  // Відстань до кінця шаблону

    6. повернути badChar
```

Ця функція створює таблицю, де для кожного символу зберігається відстань від його найправішого входження в шаблоні (крім останньої позиції) до кінця шаблону.

Попередня обробка для евристики доброго суфікса

Евристика доброго суфікса потребує складнішої попередньої обробки. Спочатку обчислюється допоміжний масив суфіксів.

```
Алгоритм ComputeSuffixArray(P):
    Вхід:
        P - шаблон довжиною m
    Вихід:
        suffix - масив довжин суфіксів

    1. m ← довжина(P)
```

```
    2. suffix ← новий масив[m]
    3. suffix[m-1] ← m

    4. для i від m-2 до 0 з кроком -1:
    5.     j ← i
    6.     поки j ≥ 0 і P[j] = P[m-1-(i-j)]:
    7.         j ← j - 1
    8.     suffix[i] ← i - j

    9. повернути suffix
```

Тепер можемо обчислити таблицю зсувів для доброго суфікса:

```
Алгоритм PreprocessGoodSuffix(P):
    Вхід:
        P - шаблон довжиною m
    Вихід:
        goodSuffix - таблиця зсувів доброго суфікса

    1. m ← довжина(P)
    2. suffix ← ComputeSuffixArray(P)
    3. goodSuffix ← новий масив[m]

    // Випадок 1: суфікс не зустрічається в іншому місці
    4. для i від 0 до m-1:
    5.     goodSuffix[i] ← m

    // Випадок 2: суфікс зустрічається в шаблоні
    6. j ← 0
    7. для i від m-1 до 0 з кроком -1:
    8.     якщо suffix[i] = i + 1:  // Префікс є суфіксом
    9.         поки j < m - 1 - i:
    10.            якщо goodSuffix[j] = m:
    11.                goodSuffix[j] ← m - 1 - i
    12.            j ← j + 1

    // Випадок 3: частина суфікса співпадає з префіксом
    13. для i від 0 до m-2:
    14.     goodSuffix[m - 1 - suffix[i]] ← m - 1 - i

    15. повернути goodSuffix
```

Основний алгоритм Боєра-Мура

```
Алгоритм BoyerMoore(T, P):
    Вхід:
        T - текст довжиною n
        P - шаблон довжиною m
```

```
Вихід:
    positions - список позицій входжень P у T

1. n ← довжина(T)
2. m ← довжина(P)
3. positions ← порожній список

// Перевірка граничних випадків
4. якщо m = 0 або m > n:
5.     повернути positions

// Попередня обробка
6. badChar ← PreprocessBadCharacter(P, алфавіт(T))
7. goodSuffix ← PreprocessGoodSuffix(P)

// Пошук
8. s ← 0  // Зсув шаблону відносно тексту

9. поки s ≤ n - m:
10.    j ← m - 1  // Починаємо з кінця шаблону

11.    // Порівнюємо символи справа наліво
12.    поки j ≥ 0 і P[j] = T[s + j]:
13.        j ← j - 1

14.    якщо j < 0:  // Знайдено повне співпадіння
15.        додати s до positions
16.        // Зсув для пошуку наступного входження
17.        якщо s + m < n:
18.            s ← s + goodSuffix[0]
19.        інакше:
20.            вийти з циклу
21.    інакше:  // Невідповідність на позиції j
22.        // Обчислюємо зсуви за обома евристиками
23.        bcShift ← j - badChar[T[s + j]]
24.        якщо bcShift < 1:
25.            bcShift ← 1
26.
27.        gsShift ← goodSuffix[j]
28.
29.        // Вибираємо максимальний зсув
30.        s ← s + max(bcShift, gsShift)

31. повернути positions
```

Розглянемо пошук шаблону P = “EXAMPLE” у тексті T = “HERE IS A SIMPLE EXAMPLE”.

Крок 1: Попередня обробка

Таблиця поганого символу:

```
Символ: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z (пробіл)
Зсув:   5 7 7 7 1 7 7 7 7 7 7 2 4 7 7 3 7 7 7 7 7 7 7 6 7 7 7
```

Таблиця доброго суфікса:

```
Позиція:     0 1 2 3 4 5 6
goodSuffix: 14 7 7 7 7 7 1
```

Крок 2: Пошук

Початкова позиція s = 0:

```
Текст:  H E R E   I S   A   S I M P L E   E X A M P L E
Шаблон: E X A M P L E
        ^^^^^^^
```

Порівнюємо справа наліво: - T[6] = ‘S’ ≠ P[6] = ‘E’ - badChar[‘S’] не визначено (використовуємо довжину шаблону) = 7 - bcShift = 6 - 7 = -1, коригуємо до 1 - gsShift = goodSuffix[6] = 1 - Зсув = max(1, 1) = 1

Продовжуємо аналогічно до позиції s = 17:

```
Текст:  H E R E   I S   A   S I M P L E   E X A M P L E
Шаблон:                         E X A M P L E
                                ^^^^^^^
```

Всі символи співпадають - знайдено входження на позиції 17.

Оптимізація алгоритму

Наведемо підходи до оптимізації алгоритму.

1. Для багатьох практичних застосувань достатньо використовувати лише евристику поганого символу:

```
Алгоритм SimplifiedBoyerMoore(T, P):
    Вхід: T - текст, P - шаблон
    Вихід: positions - список позицій

    1. n ← довжина(T)
    2. m ← довжина(P)
    3. badChar ← PreprocessBadCharacter(P, алфавіт(T))
    4. positions ← порожній список
    5. s ← 0

    6. поки s ≤ n - m:
    7.     j ← m - 1
```

```
    поки j ≥ 0 і P[j] = T[s + j]:
        j ← j - 1

    якщо j < 0:
        додати s до positions
        s ← s + (m якщо s + m < n інакше 1)
    інакше:
        skip ← j - badChar[T[s + j]]
        s ← s + max(1, skip)

повернути positions
```

2. Версія Хорспула

Алгоритм Хорспула є спрощенням Боєра-Мура, який використовує лише модифіковану евристику поганого символу:

```
Алгоритм Horspool(T, P):
    Вхід: T - текст, P - шаблон
    Вихід: позиції входжень

    n ← довжина(T)
    m ← довжина(P)

    // Таблиця зсувів базується на останньому символі вікна
    для кожного c в алфавіті:
        shift[c] ← m

    для i від 0 до m-2:
        shift[P[i]] ← m - 1 - i

    s ← 0
    поки s ≤ n - m:
        j ← m - 1
        поки j ≥ 0 і P[j] = T[s + j]:
            j ← j - 1

        якщо j < 0:
            знайдено входження на позиції s

        s ← s + shift[T[s + m - 1]]
```

Аналіз складності

Часова складність

- Попередня обробка: O(m + σ), де σ - розмір алфавіту

- Найкращий випадок: O(n/m) - коли на кожному кроці можна зсунути шаблон на всю його довжину

- Середній випадок: O(n) для випадкових текстів

- Найгірший випадок: O(nm) - коли текст і шаблон складаються з одного символу

Просторова складність

- O(m + σ) для зберігання таблиць попередньої обробки

Варіації для різних задач

Пошук з підстановочними символами

Модифікація для підтримки символу ‘?’ (будь-який один символ):

```
Алгоритм BoyerMooreWithWildcard(T, P):
    // Модифікуємо порівняння символів
    функція match(a, b):
        повернути (a = b) або (b = '?')

    // Решта алгоритму аналогічна, але використовує match()
```

Пошук множинних шаблонів

Для пошуку кількох шаблонів можна використати модифікацію з трі-структурою або автоматом.

Область застосування

Алгоритм Боєра-Мура та його варіації широко використовуються в:

- Текстові редактори - функції пошуку та заміни, особливо для великих файлів

- Системи grep - пошук патернів у файлах та потоках даних

- Антивірусне ПЗ - сканування файлів на наявність сигнатур вірусів

- Біоінформатика - пошук генетичних послідовностей у великих базах даних

- Мережева безпека - системи виявлення вторгнень (IDS) для аналізу трафіку

- Індексація документів - попередня обробка текстів для пошукових систем

- Компілятори - лексичний аналіз для довгих ключових слів

- Форензика - пошук доказів у великих обсягах даних

Приклади застосування

Приклад 1: Пошук у лог-файлах

Пошук помилок у великому лог-файлі сервера:

```
Шаблон: "ERROR: Database connection failed"
Текст: багатогігабайтний лог-файл

Алгоритм ефективно пропускає більшість тексту, зупиняючись лише на потенційних збігах.
```

Приклад 2: Валідація email

Перевірка наявності заборонених доменів:

```
Шаблони: ["@spam.com", "@phishing.net", "@malware.org"]
Використовується модифікація для множинного пошуку
```

Приклад 3: Аналіз ДНК

Пошук специфічного гена в геномі:

```
Шаблон: "ATCGATCGATCG" (специфічна послідовність)
Текст: повний геном організму
Особливість: алфавіт має лише 4 символи (A, T, C, G)
```

Переваги та недоліки

Переваги

- Висока швидкість на практиці - часто перевершує теоретично кращі алгоритми

- Ефективність для довгих шаблонів - чим довший шаблон, тим більші можливі зсуви

- Відмінна продуктивність для великих алфавітів - краще використання евристики поганого символу

- Субтлінійна складність в середньому - O(n/m) для типових текстів

- Простота реалізації спрощених версій - алгоритм Хорспула легко імплементувати

- Ефективність кешу - послідовний доступ до пам’яті

Недоліки

- Складність повної реалізації - евристика доброго суфікса складна для розуміння

- Велика попередня обробка - для коротких шаблонів може бути неефективною

- Погана продуктивність для малих алфавітів - наприклад, для бінарних даних

- Квадратична складність у гіршому випадку - O(nm) для патологічних входів

- Більше використання пам’яті - порівняно з КМП чи наївним алгоритмом

- Складність модифікації - важко адаптувати для нечіткого пошуку

Модифікація алгоритму Боєра-Мура для пошуку множинних шаблонів

Алгоритм Комментца-Вальтера (Commentz-Walter)

Алгоритм Комментца-Вальтера є узагальненням алгоритму Боєра-Мура для одночасного пошуку множини шаблонів. Він комбінує ідеї Боєра-Мура з трі-структурою (префіксним деревом), створюючи зворотне трі для ефективного пошуку багатьох шаблонів одночасно.

Основна ідея полягає в тому, що замість побудови звичайного дерева з префіксів шаблонів, алгоритм будує зворотне дерево із суфіксів шаблонів (читаючи їх справа наліво). Це дозволяє використовувати ідею Боєра-Мура про порівняння справа наліво для множини шаблонів.

Структура даних - зворотне дерево

```
Структура TrieNode:
    children - масив посилань на дочірні вузли (розмір = розмір алфав
іту)
    isEndOfPattern - булеве значення, чи закінчується тут якийсь шабл
он
    patternIndices - список індексів шаблонів, що закінчуються в цьом
у вузлі
    depth - глибина вузла в дереві
    failureLink - посилання невдачі (для автомата)
```

Побудова зворотного дерева

```
Алгоритм BuildReverseTrie(patterns[]):
    Вхід:
        patterns[] - масив шаблонів для пошуку
    Вихід:
        root - корінь зворотного дерева

    1. root ← новий TrieNode()
    2. root.depth ← 0

    3. для кожного patternIndex від 0 до довжина(patterns) - 1:
    4.     pattern ← patterns[patternIndex]
```

```
    current ← root
    
    // Вставляємо шаблон у зворотному порядку
    для i від довжина(pattern) - 1 до 0 з кроком -1:
        char ← pattern[i]
        
        якщо current.children[char] = null:
            current.children[char] ← новий TrieNode()
            current.children[char].depth ← current.depth + 1
        
        current ← current.children[char]
    
    // Позначаємо кінець шаблону
    current.isEndOfPattern ← true
    додати patternIndex до current.patternIndices

повернути root
```

Обчислення функцій зсуву

Для ефективного пошуку потрібно обчислити три функції зсуву, аналогічні евристикам Боєра-Мура:

1. Функція зсуву символу

```
Алгоритм ComputeCharShift(patterns[], alphabetSize):
    Вхід:
        patterns[] - масив шаблонів
        alphabetSize - розмір алфавіту
    Вихід:
        charShift - таблиця зсувів для кожного символу

    minLength ← мінімальна довжина серед всіх шаблонів
    
    // Ініціалізація: за замовчуванням зсув = minLength
    для кожного char від 0 до alphabetSize - 1:
        charShift[char] ← minLength
    
    // Обчислюємо зсуви для символів, що зустрічаються в шаблонах
    для кожного pattern в patterns:
        m ← довжина(pattern)
        
        // Обробляємо всі символи крім останнього
        для i від 0 до m - 2:
            charShift[pattern[i]] ← min(charShift[pattern[i]], m - 1 - i)
```

```
    14. повернути charShift
```

2. Функція зсуву вузла

```
Алгоритм ComputeNodeShift(root, patterns[]):
    Вхід:
        root - корінь зворотного дерева
        patterns[] - масив шаблонів
    Вихід:
        nodeShift - map від вузла до величини зсуву

    1. nodeShift ← новий Map()
    2. queue ← новий Queue()
    3.
    4. // BFS обхід трі
    5. додати root до queue
    6. nodeShift[root] ← 1
    7.
    8. поки queue не порожня:
    9.     node ← взяти з queue
    10.
    11.    для кожного child в node.children:
    12.        якщо child ≠ null:
    13.            додати child до queue
    14.
    15.            // Обчислюємо зсув для вузла
    16.            якщо child.isEndOfPattern:
    17.                // Мінімальна довжина серед шаблонів, що закін
чуються тут
    18.                minLen ← мінімум з довжин шаблонів в child.pat
ternIndices
    19.                nodeShift[child] ← minLen
    20.            інакше:
    21.                nodeShift[child] ← 1

    22. повернути nodeShift
```

3. Функція зсуву суфікса

```
Алгоритм ComputeSuffixShift(root, patterns[]):
    Вхід:
        root - корінь зворотного дерева
        patterns[] - масив шаблонів
    Вихід:
        suffixShift - функція зсуву для суфіксів

    1. suffixShift ← новий Map()
```

```
    2.
    3. // Для кожного вузла обчислюємо найкоротший шаблон,
    4. // що має суфікс, відповідний шляху до цього вузла
    5. функція computeForNode(node, path):
    6.     якщо node = null:
    7.         повернути
    8.
    9.     // Знаходимо найкоротше входження суфікса
    10.    minShift ← нескінченність
    11.
    12.    для кожного pattern в patterns:
    13.        якщо pattern закінчується на path:
    14.            shift ← довжина(pattern) - довжина(path)
    15.            minShift ← min(minShift, shift)
    16.
    17.    suffixShift[node] ← minShift
    18.
    19.    // Рекурсивно обробляємо дочірні вузли
    20.    для кожного char, child в node.children:
    21.        якщо child ≠ null:
    22.            computeForNode(child, char + path)
    23.
    24. computeForNode(root, "")
    25. повернути suffixShift
```

Основний алгоритм пошуку такий:

```
Алгоритм CommentzWalterSearch(text, patterns[]):
    Вхід:
        text - текст для пошуку довжиною n
        patterns[] - масив шаблонів
    Вихід:
        matches - список пар (позиція, індекс шаблону)

    1. n ← довжина(text)
    2. якщо n = 0 або patterns порожній:
    3.     повернути порожній список
    4.
    5. // Попередня обробка
    6. root ← BuildReverseTrie(patterns)
    7. charShift ← ComputeCharShift(patterns, розмір алфавіту)
    8. nodeShift ← ComputeNodeShift(root, patterns)
    9. suffixShift ← ComputeSuffixShift(root, patterns)
    10. minLen ← мінімальна довжина серед patterns
    11. matches ← порожній список
    12.
```

```
    // Основний цикл пошуку
    pos ← minLen - 1  // Починаємо з позиції останнього символу найкоротшого шаблону

    поки pos < n:
        node ← root
        j ← 0  // Лічильник символів, що співпали

        // Йдемо по трі, порівнюючи символи справа наліво
        поки pos - j ≥ 0 і node ≠ null:
            char ← text[pos - j]
            node ← node.children[char]

            якщо node ≠ null:
                j ← j + 1

                // Перевіряємо, чи знайшли повне співпадіння
                якщо node.isEndOfPattern:
                    для кожного patternIdx в node.patternIndices:
                        patternLen ← довжина(patterns[patternIdx])
                        якщо j ≥ patternLen:  // Повне співпадіння
                            matchPos ← pos - patternLen + 1
                            додати (matchPos, patternIdx) до matches
            інакше:
                вийти з циклу

        // Обчислюємо зсув
        shift ← 1  // Мінімальний зсув

        якщо j = 0:  // Немає співпадінь
            shift ← charShift[text[pos]]
        інакше:
            // Вибираємо максимальний зсув з трьох евристик
            lastNode ← той вузол, де зупинились

            // Евристика символу
            якщо pos + 1 < n:
                shift ← max(shift, charShift[text[pos + 1]])

            // Евристика вузла
            якщо lastNode в nodeShift:
                shift ← max(shift, nodeShift[lastNode])
```

```
54.
55.         // Евристика суфікса
56.         якщо lastNode в suffixShift:
57.             shift ← max(shift, suffixShift[lastNode] - j)
58.
59.     pos ← pos + shift
60.
61. повернути matches
```

Оптимізації та вдосконалення

1. Стиснення дерева

Для економії пам’яті можна використовувати стиснене дерево:

```
Алгоритм CompressTrieNode(node):
    1. якщо node має лише одну дочірню вершину і не є кінцем шаблону:
    2.     child ← єдина дочірня вершина
    3.     edge ← рядок символів на шляху до child
    4.
    5.     поки child має лише одну дочірню вершину і не є кінцем шаб
лону:
    6.         nextChild ← єдина дочірня вершина child
    7.         edge ← edge + символ переходу
    8.         child ← nextChild
    9.
    10.    // Замінюємо ланцюжок одним ребром
    11.    node.compressedEdge ← edge
    12.    node.compressedChild ← child
```

2. Використання хеш-таблиць

Для великих алфавітів ефективніше використовувати хеш-таблиці замість масивів:

```
Структура OptimizedTrieNode:
    children - HashMap<Character, TrieNode>  // Замість масиву
    // інші поля залишаються без змін
```

3. Ленива ініціалізація

Обчислювати функції зсуву можна лише для тих вузлів, які реально використовуються:

```
Алгоритм LazyShiftComputation(node, patterns):
    1. якщо node.shiftsComputed:
    2.     повернути
    3.
    4. // Обчислюємо зсуви для цього вузла
    5. node.nodeShift ← обчислити зсув вузла
```

```
    6. node.suffixShift ← обчислити зсув суфікса
    7. node.shiftsComputed ← true
```

Розглянемо пошук шаблонів [“ABC”, “BCD”, “ABCD”] у тексті “XABCDABC”:

Крок 1: Побудова зворотного дерева

```
Шаблони в зворотному порядку:
"CBA", "DCB", "DCBA"

Структура дерева:
    root
    ├── C
    │   └── B
    │       └── A (кінець "ABC")
    ├── D
    │   └── C
    │       └── B
    │           ├── A (кінець "ABCD")
    │           └── (кінець "BCD")
    │
```

Крок 2: Пошук

```
Текст: X A B C D A B C
Позиції: 0 1 2 3 4 5 6 7

pos = 2 (minLen - 1 = 3 - 1):
- Перевіряємо з позиції 2 справа наліво
- text[2] = 'B' - немає в корені дерева
- Зсув за charShift['B']

pos = 3:
- text[3] = 'C' → є в трі
- text[2] = 'B' → є продовження C→B
- text[1] = 'A' → є продовження B→A
- Знайдено "ABC" на позиції 1

І так далі...
```

Аналіз складності

- Попередня обробка: O(Σ|Pi|), де |Pi| - довжина i-го шаблону

- Пошук: O(n × m) у найгіршому випадку, але O(n/m) в середньому

- Простір: O(Σ|Pi|) для зберігання трі та таблиць

Переваги над простим Боєром-Муром

- Одночасний пошук багатьох шаблонів

- Ефективніше для великої кількості шаблонів

- Використання спільних суфіксів шаблонів

- Кращі зсуви за рахунок інформації про всі шаблони

Ця модифікація особливо ефективна для систем виявлення вторгнень, антивірусів та інших застосувань, де потрібно шукати тисячі шаблонів одночасно.

Контрольні запитання

1. В чому полягає основна ідея алгоритму Боєра-Мура, що відрізняє його від інших алгоритмів пошуку?

2. Поясніть принцип роботи евристики поганого символу на конкретному прикладі.

3. Як працює евристика доброго суфікса і чому вона складніша за евристику поганого символу?

4. Чому алгоритм Боєра-Мура особливо ефективний для довгих шаблонів?

5. В яких випадках алгоритм має найгіршу часову складність O(nm)?

6. Як обчислюється таблиця зсувів для евристики поганого символу?

7. Чим відрізняється алгоритм Хорспула від класичного Боєра-Мура?

8. Чому порівняння символів відбувається справа наліво?

9. Як вибирається фінальний зсув при конфлікті між двома евристиками?

10. Для яких типів даних алгоритм Боєра-Мура менш ефективний?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму пошуку підрядка Боєра-Мура та його модифікацій

2. Мета

Вивчити принципи роботи алгоритму Боєра-Мура, реалізувати його повну версію та спрощені варіанти, дослідити ефективність на різних типах даних та порівняти з іншими алгоритмами пошуку.

3. Завдання

- Реалізувати функції попередньої обробки для обох евристик

- Реалізувати повний алгоритм Боєра-Мура

- Реалізувати спрощену версію з однією евристикою

- Реалізувати алгоритм Хорспула

- Створити генератор тестових даних для різних сценаріїв

- Провести порівняльний аналіз з КМП та наївним алгоритмом

- Візуалізувати процес пошуку з демонстрацією зсувів

4. Опис

Лабораторна робота передбачає глибоке вивчення алгоритму Боєра-Мура через практичну реалізацію. Студенти мають реалізувати не лише базовий алгоритм, але й його оптимізації, дослідити вплив різних евристик на продуктивність та створити візуальну демонстрацію роботи алгоритму. Особлива увага приділяється аналізу ефективності на різних типах даних: випадкових текстах, природній мові, ДНК послідовностях та бінарних даних.

5. Порядок виконання

- Створити модуль BoyerMooreSearch з базовими структурами даних

- Реалізувати функцію preprocessBadCharacter для евристики поганого символу

- Реалізувати функції computeSuffixArray та preprocessGoodSuffix

- Імплементувати основну функцію пошуку boyerMooreSearch

- Додати спрощені версії: simplifiedBM та horspool

- Створити модуль тестування з різними типами даних

- Реалізувати візуалізацію кроків алгоритму

- Провести бенчмаркінг та аналіз результатів

6. Результат

Програма повинна: - Коректно знаходити всі входження шаблону в тексті - Демонструвати кроки виконання з показом зсувів - Порівнювати ефективність різних версій алгоритму - Показувати статистику порівнянь та зсувів - Візуалізувати таблиці попередньої обробки - Генерувати звіт з аналізом продуктивності

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Модуль попередньої обробки з функціями preprocessBadCharacter та preprocessGoodSuffix

- Функція основного алгоритму boyerMooreSearch

- Реалізації спрощених версій (simplified, Horspool)
- Модуль візуалізації з покроковою демонстрацією
- Модуль тестування та бенчмаркінгу
- Генератор тестових даних для різних сценаріїв

2. Продемонструвати роботу додатку:
- Покроковий пошук з візуалізацією зсувів
- Порівняння таблиць евристик для різних шаблонів
- Робота з різними типами даних (текст, ДНК, бінарні)
- Аналіз найкращого та найгіршого випадків
- Порівняльні графіки продуктивності
- Демонстрація переваг для довгих шаблонів

3. Бути готовим відповісти на питання:
- Як обчислюються зсуви для конкретних прикладів?
- Чому вибирається максимальний зсув з двох евристик?
- Як модифікувати алгоритм для пошуку з ігноруванням регістру?
- В яких випадках спрощені версії ефективніші за повну?
- Як оптимізувати алгоритм для конкретного типу даних?

Тести

1. Основна особливість алгоритму Боєра-Мура полягає в:

1) Використанні хеш-функцій

2) Порівнянні символів справа наліво

3) Побудові префікс-функції

4) Використанні суфіксного дерева

2. Евристика поганого символу дозволяє:

1) Знайти найкоротший шаблон

2) Визначити зсув на основі невідповідного символу

3) Побудувати автомат

4) Зменшити використання пам’яті

3. В найкращому випадку складність алгоритму Боєра-Мура становить:

1) O(n)

2) O(m)

3) O(n/m)

4) O(nm)

4. Алгоритм Хорспула є:

1) Розширенням Боєра-Мура

2) Спрощенням з однією евристикою

3) Незалежним алгоритмом

4) Версією для малих алфавітів

5. Для якого типу даних алгоритм Боєра-Мура найменш ефективний?

1) Природна мова

2) ДНК послідовності

3) Випадковий текст

4) Бінарні дані

Відповіді:

1. b)

2. b)

3. c)

4. b)

5. d)

Тести підвищеної складності

1. При пошуку шаблону “AAAA” в тексті “AAAAAAAAAA” скільки порівнянь символів виконає алгоритм Боєра-Мура?

1) 10

2) 16

3) 19

4) 40

2. Яка максимальна величина зсуву можлива для шаблону довжини m?

1) m

2) m-1

3) m+1

4) 2m

3. Для шаблону “ABCDEF” і невідповідності на символі ‘C’ в позиції j=2, який зсув дасть евристика доброго суфікса, якщо суфікс “DEF” не зустрічається в іншому місці?

1) 1

2) 3

3) 6

4) 4

4. При якій умові евристика поганого символу дає від’ємний зсув?

1) Коли символ не зустрічається в шаблоні

2) Коли символ зустрічається правіше поточної позиції

3) Коли шаблон складається з одного символу

4) Ніколи

5. Яка просторова складність попередньої обробки для алфавіту розміром σ?

1) O(m)

2) O(σ)

3) O(m + σ)

4) O(m × σ)

6. Для оптимізації кешу процесора краще використовувати:

1) Повний алгоритм Боєра-Мура

2) Алгоритм Хорспула

3) КМП

4) Наївний алгоритм

7. При пошуку в тексті з рівномірним розподілом 256 символів ASCII, середня кількість порівнянь на позицію для шаблону довжини 10 становить приблизно:

1) 10

2) 5

3) 1

4) 0.04

8. Яка модифікація алгоритму найефективніша для пошуку в стиснутих даних?

1) Класичний Боєра-Мура

2) Турбо Боєра-Мура

3) Адаптований для бітових операцій

4) Алгоритм не підходить для стиснутих даних

9. При використанні лише евристики поганого символу для шаблону з унікальними символами, мінімальний зсув (крім 1) становить:

1) 2

2) m/2

3) m-1

4) Залежить від алфавіту

10. Для ефективного пошуку множини з k шаблонів краще використовувати:

1) k окремих виконань Боєра-Мура

2) Модифікацію з трі-структурою

3) Алгоритм Ахо-Корасік

4) Паралельні потоки Боєра-Мура

Відповіді:

1. c) Патологічний випадок з багатьма частковими співпадіннями

2. a) Максимальний зсув дорівнює довжині шаблону

3. c) Зсув на всю довжину шаблону, якщо суфікс унікальний

4. b) Коли невідповідний символ є в шаблоні правіше поточної позиції

5. c) Масив суфіксів O(m) + таблиця символів O(σ)

6. b) Хорспул має простішу логіку та кращу локальність даних

7. d) Приблизно 10/256 для рівномірного розподілу

8. d) Стиснуті дані не мають посимвольної структури

9. a) Мінімальний зсув для унікальних символів

10. c) Ахо-Корасік спеціально розроблений для множинного пошуку

Завдання для самостійної роботи

1. Реалізуйте турбо-версію алгоритму Боєра-Мура, яка запам’ятовує інформацію про попередні порівняння для уникнення повторних перевірок.

2. Створіть адаптивний алгоритм, який автоматично вибирає між Боєром-Муром, КМП та наївним пошуком залежно від характеристик вхідних даних.

3. Розробіть версію для пошуку з обмеженою кількістю невідповідностей (approximate matching) на основі Боєра-Мура.

4. Імплементуйте двовимірну версію алгоритму для пошуку прямокутних патернів у матрицях.

5. Створіть оптимізовану версію для пошуку в Unicode-текстах з правильною обробкою багатобайтових символів.

6. Реалізуйте паралельну версію алгоритму з розподілом тексту між потоками та синхронізацією результатів.

7. Розробіть модифікацію для потокового пошуку в даних, що надходять частинами, з мінімальною буферизацією.

8. Створіть гібридний алгоритм, який комбінує Боєра-Мура з бітовими операціями для малих алфавітів.

9. Імплементуйте версію з підтримкою регулярних виразів обмеженої складності (символи підстановки, класи символів).

10. Розробіть систему автоматичного вибору оптимальних параметрів алгоритму на основі машинного навчання.

Теми лабораторних робіт

1. Розробка високопродуктивної системи пошуку для текстового редактора з підтримкою великих файлів та інкрементального пошуку.

2. Створення антивірусного сканера на основі Боєра-Мура для виявлення сигнатур шкідливого ПЗ у виконуваних файлах.

3. Імплементація системи моніторингу мережевого трафіку з використанням модифікованого Боєра-Мура для виявлення атак.

4. Розробка біоінформатичного інструменту для пошуку генів у геномних послідовностях з візуалізацією результатів.

5. Створення системи детекції плагіату в програмному коді з урахуванням синтаксичних особливостей мов програмування.

6. Реалізація пошукової системи для корпусу документів з індексацією на основі алгоритму Боєра-Мура.

7. Розробка інструменту форензичного аналізу для пошуку доказів у образах дисків та дампах пам’яті.

8. Створення оптимізованої бібліотеки пошуку для вбудованих систем з обмеженими ресурсами.

9. Імплементація системи фільтрації контенту реального часу для веб-проксі сервера.

10. Розробка крос-платформної утиліти grep з розширеними можливостями на основі різних варіантів Боєра-Мура.

10.2.6. Алгоритм пошуку підрядка Ахо-Корасік

Теоретичні відомості

Алгоритм Ахо-Корасік, розроблений Альфредом Ахо та Маргарет Корасік у 1975 році, є потужним алгоритмом для одночасного пошуку множини шаблонів у тексті. Він об’єднує концепції префіксного дерева та скінченного автомата, створюючи ефективну структуру даних для розпізнавання багатьох патернів за один прохід по тексту.

Основна перевага алгоритму полягає в тому, що час пошуку не залежить від кількості шаблонів. Незалежно від того, шукаємо ми 10 чи 10000 шаблонів, текст обробляється лише один раз. Це робить алгоритм надзвичайно ефективним для задач, де потрібно знайти багато різних патернів одночасно.

Основні компоненти алгоритму

Алгоритм Ахо-Корасік складається з трьох основних компонентів:

1. Префіксне дерево (Trі) - структура для зберігання всіх шаблонів

2. Функція невдачі (Failure Function) - визначає, куди переходити при невідповідності символів

3. Функція виходу (Output Function) - вказує, які шаблони знайдено в кожному стані

Структура даних

Кожен вузол автомата Ахо-Корасік містить таку інформацію:

```
Структура AhoCorasickNode:
    children - масив або словник дочірніх вузлів для кожного символу
    parent - посилання на батьківський вузол
    charFromParent - символ, по якому прийшли з батька
    isEndOfPattern - чи закінчується в цьому вузлі якийсь шаблон
    patternIndices - список індексів шаблонів, що закінчуються тут
    failureLink - посилання невдачі
    outputLink - посилання на найближчий вузол з непорожнім виходом
    depth - глибина вузла в дереві
```

Фаза 1: Побудова префіксного дерева

Першим кроком є побудова трі-структури для всіх шаблонів. Це дерево дозволяє ефективно зберігати та шукати префікси шаблонів.

```
Алгоритм BuildTrie(patterns[]):
    Вхід:
        patterns[] - масив шаблонів для пошуку
    Вихід:
        root - корінь побудованого трі

    1. root ← новий AhoCorasickNode()
    2. root.parent ← null
    3. root.depth ← 0

    4. для кожного patternIndex від 0 до довжина(patterns) - 1:
    5.     pattern ← patterns[patternIndex]
    6.     current ← root
    7.
    8.     // Вставляємо кожен символ шаблону
    9.     для кожного i від 0 до довжина(pattern) - 1:
    10.        char ← pattern[i]
    11.
    12.        // Якщо немає переходу по цьому символу, створюємо нов
ий вузол
    13.        якщо current.children[char] = null:
    14.            newNode ← новий AhoCorasickNode()
    15.            newNode.parent ← current
    16.            newNode.charFromParent ← char
    17.            newNode.depth ← current.depth + 1
    18.            current.children[char] ← newNode
    19.
    20.        current ← current.children[char]
    21.
    22.    // Позначаємо кінець шаблону
    23.    current.isEndOfPattern ← true
    24.    додати patternIndex до current.patternIndices
```

```
    25. повернути root
```

Фаза 2: Побудова функції невдачі

Функція невдачі визначає, куди переходити, коли немає прямого переходу з поточного стану по символу, що читається. Вона вказує на найдовший власний суфікс рядка, що відповідає шляху від кореня до поточного вузла, який також є префіксом якогось шаблону.

```
Алгоритм BuildFailureFunction(root):
    Вхід:
        root - корінь трі з шаблонами
    Вихід:
        оновлене дерево з failure-посиланнями

    1. queue ← новий Queue()
    2.
    3. // Ініціалізація: failure-посилання для дочірніх вузлів кореня
    4. для кожного char, child в root.children:
    5.     якщо child ≠ null:
    6.         child.failureLink ← root
    7.         додати child до queue
    8.
    9. // Failure-посилання для кореня вказує на себе
    10. root.failureLink ← root
    11.
    12. // BFS для побудови failure-посилань для решти вузлів
    13. поки queue не порожня:
    14.     current ← взяти з queue
    15.
    16.     // Обробляємо всіх дочірніх вузлів
    17.     для кожного char, child в current.children:
    18.         якщо child ≠ null:
    19.             додати child до queue
    20.
    21.             // Шукаємо failure-посилання для child
    22.             failNode ← current.failureLink
    23.
    24.             // Йдемо по failure-посиланнях, поки не знайдемо
перехід
    25.             поки failNode ≠ root і failNode.children[char] =
null:
    26.                 failNode ← failNode.failureLink
    27.
    28.             // Встановлюємо failure-посилання
```

```
    29.             якщо failNode.children[char] ≠ null і failNode.ch
ildren[char] ≠ child:
    30.                 child.failureLink ← failNode.children[char]
    31.             інакше:
    32.                 child.failureLink ← root
    33.
    34.             // Об'єднуємо виходи з failure-вузла
    35.             якщо child.failureLink.isEndOfPattern:
    36.                 додати всі патерни з child.failureLink.patter
nIndices
    37.                 до child.patternIndices
```

Фаза 3: Побудова функції виходу та оптимізація

Для ефективного визначення всіх шаблонів, що закінчуються в даному стані, будуємо додаткові output-посилання:

```
Алгоритм BuildOutputFunction(root):
    Вхід:
        root - корінь трі з failure-посиланнями
    Вихід:
        оновлене дерево з output-посиланнями

    1. queue ← новий Queue()
    2. додати root до queue
    3. root.outputLink ← null
    4.
    5. поки queue не порожня:
    6.     current ← взяти з queue
    7.
    8.     для кожного child в current.children:
    9.         якщо child ≠ null:
    10.            додати child до queue
    11.
    12.            // Встановлюємо output-посилання
    13.            якщо child.isEndOfPattern:
    14.                child.outputLink ← child
    15.            інакше якщо child.failureLink ≠ null:
    16.                child.outputLink ← child.failureLink.outputLin
k
    17.            інакше:
    18.                child.outputLink ← null
```

Фаза 4: Пошук шаблонів у тексті

Після побудови автомата можемо ефективно шукати всі входження шаблонів:

```
Алгоритм AhoCorasickSearch(text, root):
    Вхід:
        text - текст для пошуку довжиною n
        root - корінь автомата Ахо-Корасік
    Вихід:
        matches - список пар (позиція, індекс шаблону)

    1. n ← довжина(text)
    2. matches ← порожній список
    3. current ← root
    4.
    5. для i від 0 до n - 1:
    6.     char ← text[i]
    7.
    8.     // Знаходимо наступний стан автомата
    9.     поки current ≠ root і current.children[char] = null:
    10.        current ← current.failureLink
    11.
    12.    якщо current.children[char] ≠ null:
    13.        current ← current.children[char]
    14.
    15.    // Збираємо всі знайдені шаблони в поточному стані
    16.    temp ← current
    17.    поки temp ≠ null:
    18.        якщо temp.isEndOfPattern:
    19.            для кожного patternIdx в temp.patternIndices:
    20.                // Позиція початку = поточна позиція - довжина
шаблону + 1
    21.                startPos ← i - patterns[patternIdx].length + 1
    22.                додати (startPos, patternIdx) до matches
    23.
    24.        // Переходимо по output-посиланню для пошуку інших шаб
лонів
    25.        temp ← temp.outputLink
    26.        якщо temp = current:  // Уникаємо нескінченного циклу
    27.            вийти з циклу

    28. повернути matches
```

Розглянемо пошук шаблонів [“he”, “she”, “his”, “hers”] у тексті “ushers”.

Крок 1: Побудова дерева

```
Будуємо дерево для шаблонів:
    root
    ├── h
    │   ├── e (кінець "he")
```

```
    │   │   └── r
    │   │       └── s (кінець "hers")
    │   └── i
    │       └── s (кінець "his")
    └── s
        └── h
            └── e (кінець "she")
```

Крок 2: Побудова failure-посилань

Для кожного вузла визначаємо failure-посилання: - Вузол ‘h’ → root (немає префікса ‘h’) - Вузол ‘he’ → вузол ‘e’ (немає окремого вузла ‘e’, тому root) - Вузол ‘s’ → root - Вузол ‘sh’ → вузол ‘h’ - Вузол ‘she’ → вузол ‘he’ - І так далі…

Крок 3: Процес пошуку в тексті “ushers”

```
Позиція 0, символ 'u':
- current = root
- root не має переходу по 'u'
- Залишаємось в root

Позиція 1, символ 's':
- current = root
- Переходимо: root → s
- current = вузол 's'

Позиція 2, символ 'h':
- current = вузол 's'
- Переходимо: s → h
- current = вузол 'sh'

Позиція 3, символ 'e':
- current = вузол 'sh'
- Переходимо: sh → e
- current = вузол 'she' (кінець шаблону!)
- Знайдено "she" на позиції 1
- По failure-посиланню також знаходимо "he" на позиції 2

Позиція 4, символ 'r':
- current = вузол 'she'
- Немає прямого переходу по 'r'
- Йдемо по failure: she → he
- he має перехід по 'r': he → r
- current = вузол 'her'

Позиція 5, символ 's':
- current = вузол 'her'
```

```
- Переходимо: her → s
- current = вузол 'hers' (кінець шаблону!)
- Знайдено "hers" на позиції 2
```

Оптимізації алгоритму

1. Стиснення переходів

Замість зберігання масиву для всіх можливих символів, використовуємо хеш-таблицю:

```
Алгоритм GetNextState(current, char):
    1. поки current ≠ null:
    2.     якщо current.children містить char:
    3.         повернути current.children[char]
    4.     якщо current = root:
    5.         повернути root
    6.     current ← current.failureLink
    7. повернути root
```

2. Попереднє обчислення переходів

Для прискорення пошуку можна попередньо обчислити всі переходи:

```
Алгоритм PrecomputeTransitions(root, alphabet):
    1. для кожного state в всіх вузлах автомата:
    2.     для кожного char в alphabet:
    3.         якщо state.children[char] існує:
    4.             transition[state][char] ← state.children[char]
    5.         інакше:
    6.             // Обчислюємо перехід через failure-посилання
    7.             temp ← state
    8.             поки temp ≠ root і temp.children[char] не існує:
    9.                 temp ← temp.failureLink
    10.            якщо temp.children[char] існує:
    11.                transition[state][char] ← temp.children[char]
    12.            інакше:
    13.                transition[state][char] ← root
```

Аналіз складності

Часова складність:

- Побудова автомата: O(M), де M - сумарна довжина всіх шаблонів

- Пошук: O(n + z), де n - довжина тексту, z - кількість знайдених входжень

- Загальна: O(M + n + z)

Просторова складність:

- O(M × |Σ|) у найгіршому випадку, де |Σ| - розмір алфавіту

- O(M) при використанні хеш-таблиць для переходів

Модифікації для спеціальних задач

1. Пошук з врахуванням регістру

```
Алгоритм CaseInsensitiveSearch(text, patterns):
    1. // Перетворюємо всі шаблони на нижній регістр
    2. lowerPatterns ← []
    3. для кожного pattern в patterns:
    4.   додати toLowerCase(pattern) до lowerPatterns
    5.
    6. // Будуємо автомат з перетвореними шаблонами
    7. root ← BuildAhoCorasick(lowerPatterns)
    8.
    9. // Пошук з перетворенням символів
    10. matches ← []
    11. current ← root
    12. для і від 0 до довжина(text) - 1:
    13.   char ← toLowerCase(text[i])
    14.   // Стандартний пошук з char
```

2. Пошук з підстановочними символами

```
Алгоритм WildcardSearch(text, patternsWithWildcards):
    1. // Розбиваємо кожен шаблон на сегменти між wildcard'ами
    2. segments ← []
    3. для кожного pattern в patternsWithWildcards:
    4.     parts ← split(pattern, '?')  // '?' - wildcard
    5.     додати parts до segments
    6.
    7. // Будуємо автомат для всіх сегментів
    8. // Потім перевіряємо відстані між знайденими сегментами
```

Область застосування

Алгоритм Ахо-Корасік широко використовується в:

- Системи виявлення вторгнень (IDS) - сканування мережевого трафіку на наявність тисяч сигнатур атак

- Антивірусне програмне забезпечення - пошук сигнатур вірусів у файлах

- Біоінформатика - пошук множини генетичних послідовностей у геномах

- Фільтрація контенту - виявлення заборонених слів або фраз у текстах

- Пошукові системи - знаходження множини ключових слів у документах

- Обробка природної мови - виявлення іменованих сутностей

- Компілятори - лексичний аналіз для розпізнавання ключових слів

- Системи моніторингу - пошук критичних повідомлень у логах

Приклади застосування

Приклад 1: Фільтрація спаму

Система фільтрації електронної пошти використовує Ахо-Корасік для виявлення спам-фраз:

```
Шаблони: ["win money", "click here", "limited offer", "act now"]
Текст листа: "Click here for limited offer! Act now to win money!"
Результат: знайдено 4 спам-індикатори
```

Приклад 2: Аналіз ДНК

Пошук множини генетичних маркерів у послідовності ДНК:

```
Маркери: ["ATCG", "CGTA", "TAGA"]
ДНК: "GATCGTAGATCGTAGA"
Знайдено: ATCG на позиції 1, CGTA на позиціях 3 та 10, TAGA на позиці
ї 12
```

Приклад 3: Моніторинг безпеки

Виявлення підозрілих команд у системних логах:

```
Небезпечні команди: ["rm -rf", "sudo passwd", "chmod 777"]
Аналіз логів в реальному часі для виявлення потенційних загроз
```

Переваги та недоліки

Переваги

- Лінійна складність пошуку - O(n) незалежно від кількості шаблонів

- Одночасний пошук багатьох шаблонів - ефективність не падає зі збільшенням кількості

- Детерміністична поведінка - гарантована продуктивність

- Знаходить всі входження - включаючи ті, що перекриваються

- Ефективне використання спільних префіксів - економія пам’яті

- Потокова обробка - не потребує зберігання всього тексту

Недоліки

- Значне використання пам’яті - особливо для великих алфавітів

- Складність реалізації - порівняно з простими алгоритмами

- Час попередньої обробки - побудова автомата може бути повільною для великої кількості довгих шаблонів

- Не оптимальний для одного шаблону - КМП або Боєр-Мур ефективніші

- Фіксований набір шаблонів - додавання нових потребує перебудови

- Чутливість до регістру - потребує модифікації для ігнорування регістру

Контрольні запитання

1. Які три основні компоненти складають алгоритм Ахо-Корасік?

2. Що таке failure-посилання і яка його роль в алгоритмі?

3. Чому алгоритм ефективний для пошуку багатьох шаблонів одночасно?

4. Як працює функція виходу (output function)?

5. Яка часова складність пошуку і від чого вона залежить?

6. Чим відрізняється автомат Ахо-Корасік від звичайного префіксного дерева?

7. Як обробляються шаблони, що перекриваються?

8. Для яких задач алгоритм Ахо-Корасік неефективний?

9. Як модифікувати алгоритм для пошуку з ігноруванням регістру?

10. Що таке output-посилання і навіщо воно потрібне?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму пошуку множинних шаблонів Ахо-Корасік

2. Мета

Вивчити принципи побудови та роботи автомата Ахо-Корасік, реалізувати повний алгоритм з усіма оптимізаціями, дослідити його ефективність для різних задач пошуку множинних шаблонів.

3. Завдання

- Реалізувати структуру даних для вузлів автомата

- Реалізувати побудову префіксного дерева

- Реалізувати алгоритм побудови failure-функції

- Реалізувати алгоритм побудови output-функції

- Створити ефективний алгоритм пошуку

- Додати візуалізацію автомата

- Провести порівняння з послідовним пошуком окремих шаблонів

- Реалізувати модифікації для спеціальних випадків

4. Опис

Лабораторна робота передбачає повну реалізацію алгоритму Ахо-Корасік від базових структур даних до оптимізованих версій. Студенти мають не лише реалізувати алгоритм, але й дослідити його поведінку на різних типах даних, створити візуалізацію для кращого розуміння роботи автомата та порівняти ефективність з іншими підходами до пошуку множинних шаблонів.

5. Порядок виконання

- Створити клас AhoCorasickNode для представлення вузлів

- Реалізувати клас AhoCorasickAutomaton з основними методами

- Імплементувати метод buildTrie для побудови префіксного дерева

- Реалізувати метод buildFailureLinks використовуючи BFS

- Додати метод buildOutputLinks для оптимізації виходу

- Створити метод search для пошуку всіх входжень

- Реалізувати візуалізацію автомата (граф з failure-посиланнями)

- Додати методи для статистики та аналізу

- Створити тестові сценарії для різних випадків використання

- Реалізувати модифікації (регістронезалежний пошук, тощо)

6. Результат

Готова система повинна: - Коректно будувати автомат для довільного набору шаблонів - Знаходити всі входження всіх шаблонів за один прохід - Візуалізувати структуру автомата - Показувати статистику: кількість станів, переходів, знайдених входжень - Демонструвати покроковий процес пошуку - Порівнювати продуктивність з альтернативними підходами

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас AhoCorasickNode з усіма необхідними полями

- Клас AhoCorasickAutomaton з методами побудови

- Метод buildTrie для створення префіксного дерева

- Метод buildFailureFunction з BFS обходом
- Метод buildOutputFunction для оптимізації
- Метод search з детальними коментарями
- Модуль візуалізації автомата
- Модуль тестування та бенчмаркінгу

2. Продемонструвати роботу додатку:
- Побудова автомата для набору шаблонів
- Візуалізація структури з failure та output посиланнями
- Покроковий пошук з показом поточного стану
- Знаходження перекриваючихся входжень
- Порівняння часу виконання для різної кількості шаблонів
- Робота модифікацій (регістронезалежний пошук, тощо)

3. Бути готовим відповісти на питання:
- Як обчислюються failure-посилання для конкретних вузлів?
- Чому використовується BFS при побудові failure-функції?
- Як додати новий шаблон до існуючого автомата?
- Яка різниця між failure та output посиланнями?
- Як оптимізувати автомат для конкретного алфавіту?

Тести

1. Основна перевага алгоритму Ахо-Корасік полягає в:

1) Простоті реалізації

2) Мінімальному використанні пам’яті

3) Пошуку багатьох шаблонів за один прохід

4) Швидкості для одного шаблону

2. Failure-посилання вказує на:

1) Попередній вузол в дереві

2) Найдовший суфікс, який є префіксом

3) Наступний символ шаблону

4) Корінь дерева

3. Часова складність пошуку в алгоритмі Ахо-Корасік:

1) O(n × m)

2) O(n + m)

3) O(n + z)

4) O(n log m)

4. Output-посилання використовується для:

1) Побудови дерева

2) Знаходження всіх шаблонів у стані

3) Переходу між станами

4) Зберігання результатів

5. При побудові failure-функції використовується:

1) DFS (пошук в глибину)

2) BFS (пошук в ширину)

3) Динамічне програмування

4) Жадібний алгоритм

Відповіді:

1. c)

2. b)

3. c)

4. b)

5. b)

Тести підвищеної складності

1. Скільки failure-посилань може вказувати на один вузол v на глибині d?

1) Не більше d

2) Не більше 2^d

3) Необмежено

4) Рівно d-1

2. Для шаблонів {“he”, “she”, “his”, “hers”} скільки станів матиме автомат (включаючи корінь)?

1) 9

2) 10

3) 11

4) 12

3. При додаванні нового шаблону до існуючого автомата потрібно:

1) Повністю перебудувати автомат

2) Оновити лише префіксне дерево

3) Оновити дерево та перерахувати всі failure-посилання

4) Додати лише нові вузли

4. Максимальна кількість переходів по failure-посиланням при обробці одного символу:

1) O(m), де m - довжина найдовшого шаблону

2) O(log m)

3) O(1) амортизовано

4) O(k), де k - кількість шаблонів

5. Для алфавіту розміру σ та M символів у всіх шаблонах, просторова складність з масивами:

1) O(M)

2) O(M × σ)

3) O(M + σ)

4) O(σ^M)

6. При пошуку в тексті “аааа” шаблонів {“аа”, “ааа”} скільки входжень буде знайдено?

1) 2

2) 3

3) 4

4) 5

7. Яка структура даних найефективніша для зберігання переходів при великому алфавіті?

1) Масив

2) Хеш-таблиця

3) Збалансоване дерево

4) Список

8. Глибина failure-посилання завжди:

1) Менша за глибину поточного вузла

2) Більша за глибину поточного вузла

3) Дорівнює глибині поточного вузла

4) Не залежить від глибини

9. Для оптимізації кеш-пам’яті краще:

1) Зберігати всі переходи в масиві

2) Використовувати вказівники для переходів

3) Попередньо обчислити всі можливі переходи

4) Використовувати стиснене представлення

10. При модифікації для пошуку з помилками (approximate matching) найскладніше:

1) Побудувати дерево

2) Обчислити failure-посилання

3) Визначити стани з частковим співпадінням

4) Знайти всі входження

Відповіді:

1. c) Необмежено - багато вузлів можуть мати однаковий найдовший суфікс

2. b) 10 - корінь + 9 вузлів для символів всіх шаблонів

3. c) - Потрібно оновити дерево та перерахувати failure/output посилання

4. a) O(m) - в гіршому випадку проходимо всі failure-посилання до кореня

5. b) O(M × σ) - кожен вузол може мати σ переходів

6. d) 5 - “aa” на позиціях 0,1,2 та “aaa” на позиціях 0,1

7. b) - Хеш-таблиця ефективна для розріджених переходів

8. a) - Failure-посилання завжди веде на меншу глибину

9. c) - Попереднє обчислення покращує локальність даних

10. c) - Потрібно відстежувати кількість помилок для кожного стану

Завдання для самостійної роботи

1. Реалізуйте динамічну версію алгоритму Ахо-Корасік, яка дозволяє додавати та видаляти шаблони без повної перебудови автомата.

2. Створіть паралельну версію алгоритму для обробки великих текстів на багатоядерних системах з ефективним розподілом навантаження.

3. Розробіть модифікацію для приблизного пошуку з підтримкою заданої кількості помилок (вставки, видалення, заміни).

4. Імплементуйте версію з підтримкою регулярних виразів обмеженого типу (класи символів, квантифікатори).

5. Створіть адаптивну версію, яка автоматично вибирає між Ахо-Корасік та іншими алгоритмами залежно від характеристик задачі.

6. Реалізуйте двовимірний варіант для пошуку прямокутних патернів у матрицях символів.

7. Розробіть стиснене представлення автомата для зменшення використання пам'яті при збереженні швидкодії.

8. Створіть версію для потокової обробки з обмеженим буфером для роботи з нескінченними потоками даних.

9. Імплементуйте гібридний алгоритм, що комбінує Ахо-Корасік з хешуванням для великих наборів коротких шаблонів.

10. Розробіть версію з підтримкою Unicode та правильною обробкою різних кодувань тексту.

Теми лабораторних робіт

1. Розробка системи виявлення вторгнень (IDS) на основі Ахо-Корасік для аналізу мережевого трафіку в реальному часі.

2. Створення антивірусного движка для сканування файлів на наявність тисяч сигнатур шкідливого ПЗ.

3. Імплементація системи моніторингу соціальних мереж для виявлення тенденцій та небезпечного контенту.

4. Розробка біоінформатичного інструменту для одночасного пошуку множини генетичних маркерів у геномах.

5. Створення системи фільтрації контенту для освітніх закладів з можливістю налаштування категорій.

6. Реалізація пошукової системи для корпоративної бази знань з підтримкою складних запитів.

7. Розробка інструменту для виявлення плагіату в наукових роботах з аналізом перефразувань.

8. Створення системи аналізу логів для DevOps з виявленням критичних подій та аномалій.

9. Імплементація real-time системи модерації чатів та форумів з автоматичним виявленням порушень.

10. Розробка лінгвістичного процесора для виявлення та класифікації іменованих сутностей у текстах.

10.2.7. Алгоритм пошуку підрядка: Z-алгоритм

Теоретичні відомості

Z-алгоритм є потужним та елегантним алгоритмом для обробки рядків, який обчислює для кожної позиції рядка довжину найдовшого підрядка, що починається з цієї позиції і збігається з префіксом всього рядка. Незважаючи на свою простоту, алгоритм має лінійну часову складність O(n) і широко застосовується для розв'язання різноманітних задач пошуку підрядків та аналізу рядків.

Алгоритм отримав свою назву від Z-масиву (Z-array), який він обчислює. Для рядка S довжиною n, Z[i] визначається як довжина найдовшого підрядка, що починається з позиції i і збігається з префіксом S. Формально, Z[i] - це максимальне k таке, що S[0..k-1] = S[i..i+k-1].

Основні поняття та визначення

Z-масив

Z-масив для рядка S - це масив Z довжиною n, де: - Z[0] зазвичай не визначений або встановлюється рівним n (довжина всього рядка) - Z[i] для i > 0 - довжина найдовшого префікса S, який також є підрядком, що починається з позиції i

Наприклад, для рядка S = “aabaaab”:

```
Позиція i: 0 1 2 3 4 5 6
S[i]:      a a b a a a b
Z[i]:      7 1 0 2 2 1 0
```

Пояснення: - Z[1] = 1, бо S[1..1] = “a” збігається з S[0..0] = “a” - Z[2] = 0, бо S[2] = “b” не збігається з S[0] = “a” - Z[3] = 2, бо S[3..4] = “aa” збігається з S[0..1] = “aa” - Z[4] = 2, бо S[4..5] = “aa” збігається з S[0..1] = “aa”

Z-блок

Z-блок (Z-box) - це інтервал [l, r], де l та r - індекси в рядку такі, що підрядок S[l..r] збігається з префіксом S[0..r-l]. Z-блоки відіграють ключову роль в ефективному обчисленні Z-масиву.

Наївний алгоритм обчислення Z-масиву

Перед розглядом оптимізованого алгоритму, розглянемо наївний підхід:

```
Алгоритм NaiveZArray(S):
    Вхід:
        S - рядок довжиною n
    Вихід:
        Z - Z-масив для рядка S

    1. n ← довжина(S)
    2. Z ← новий масив[n]
    3. Z[0] ← n  // або залишити невизначеним

    4. для i від 1 до n-1:
    5.     k ← 0
    6.     // Порівнюємо символи починаючи з позиції i з префіксом
    7.     поки i + k < n і S[k] = S[i + k]:
    8.         k ← k + 1
    9.     Z[i] ← k

    10. повернути Z
```

Цей алгоритм має часову складність $O(n^2)$ у найгіршому випадку, що неефективно для великих рядків.

Оптимізований Z-алгоритм

Ключова ідея оптимізації полягає у використанні вже обчислених значень Z-масиву для прискорення обчислення наступних значень. Алгоритм підтримує інтервал [l, r] - найправіший Z-блок, знайдений до поточного моменту.

```
Алгоритм ZAlgorithm(S):
    Вхід:
        S - рядок довжиною n
    Вихід:
        Z - Z-масив для рядка S

    1. n ← довжина(S)
    2. Z ← новий масив[n]
    3. Z[0] ← n
    4. l ← 0, r ← 0  // Межі найправішого Z-блоку
```

```
5. для i від 1 до n-1:
6.     // Випадок 1: i > r (поза поточним Z-блоком)
7.     якщо i > r:
8.         l ← i, r ← i
9.         // Розширюємо Z-блок наскільки можливо
10.        поки r < n і S[r-l] = S[r]:
11.            r ← r + 1
12.        Z[i] ← r - l
13.        r ← r - 1
14.
15.    // Випадок 2: i ≤ r (всередині поточного Z-блоку)
16.    інакше:
17.        k ← i - l  // Відповідна позиція в префіксі
18.
19.        // Випадок 2a: Z[k] < r - i + 1
20.        якщо Z[k] < r - i + 1:
21.            Z[i] ← Z[k]
22.
23.        // Випадок 2b: Z[k] ≥ r - i + 1
24.        інакше:
25.            l ← i
26.            // Намагаємось розширити Z-блок за межі r
27.            поки r < n і S[r-l] = S[r]:
28.                r ← r + 1
29.            Z[i] ← r - l
30.            r ← r - 1

31. повернути Z
```

Розглянемо детальне пояснення кроків алгоритму:

Ініціалізація (кроки 1-4)

Створюємо Z-масив та ініціалізуємо змінні l та r, які відстежують межі найправішого Z-блоку. Спочатку немає жодного Z-блоку, тому l = r = 0.

Основний цикл (кроки 5-30)

Для кожної позиції i від 1 до n-1 розглядаємо два основні випадки:

Випадок 1: i > r (позиція i знаходиться за межами поточного Z-блоку)

У цьому випадку ми не можемо використати попередньо обчислену інформацію, тому обчислюємо Z[i] безпосередньо, порівнюючи символи починаючи з позиції i з префіксом рядка. Після обчислення оновлюємо межі Z-блоку.

Випадок 2: i ⩽ r (позиція i знаходиться всередині поточного Z-блоку)

Оскільки S[l..r] = S[0..r-l], ми знаємо, що S[i..r] = S[i-l..r-l]. Це дозволяє використати значення Z[k], де k = i - l.

- Підвипадок 2a: Якщо Z[k] < r - i + 1, то Z[i] = Z[k]. Це означає, що збіг закінчується до межі поточного Z-блоку.

- Підвипадок 2b: Якщо Z[k] ⩾ r - i + 1, збіг може продовжуватися за межі r, тому потрібно явно перевірити символи після позиції r.

Розглянемо роботу алгоритму для рядка S = “aabcaabxaab”:

```
Крок 1: i = 1
S = a a b c a a b x a a b
    ^ ^
l = 1, r = 1 (новий Z-блок)
Порівнюємо: S[0] = 'a' = S[1] = 'a' ✓
           S[1] = 'a' ≠ S[2] = 'b' ✗
Z[1] = 1, r = 1

Крок 2: i = 2
S = a a b c a a b x a a b
      ^
i > r, тому обчислюємо напряму
S[0] = 'a' ≠ S[2] = 'b' ✗
Z[2] = 0, l = 2, r = 1 (r < l, тому фактично немає Z-блоку)

Крок 3: i = 3
S = a a b c a a b x a a b
        ^
i > r, обчислюємо напряму
S[0] = 'a' ≠ S[3] = 'c' ✗
Z[3] = 0

Крок 4: i = 4
S = a a b c a a b x a a b
           ^
i > r, обчислюємо напряму
S[0] = 'a' = S[4] = 'a' ✓
S[1] = 'a' = S[5] = 'a' ✓
S[2] = 'b' = S[6] = 'b' ✓
S[3] = 'c' ≠ S[7] = 'x' ✗
Z[4] = 3, l = 4, r = 6

Крок 5: i = 5
S = a a b c a a b x a a b
             ^
```

```
i ≤ r (5 ≤ 6), k = i - l = 5 - 4 = 1
Z[k] = Z[1] = 1
r - i + 1 = 6 - 5 + 1 = 2
Z[1] < 2, тому Z[5] = Z[1] = 1

Крок 6: i = 6
S = a a b c a a b x a a b
                ^
i ≤ r (6 ≤ 6), k = 2
Z[2] = 0 < 1, тому Z[6] = 0

І так далі...

Фінальний Z-масив: [11, 1, 0, 0, 3, 1, 0, 0, 3, 1, 0]
```

Використання Z-алгоритму для пошуку підрядка

Для пошуку всіх входжень шаблону P у тексті T, створюємо об’єднаний рядок S = P + “$" + T, де "$” - спеціальний символ, якого немає ні в P, ні в T.

```
Алгоритм ZSearch(T, P):
    Вхід:
        T - текст довжиною n
        P - шаблон довжиною m
    Вихід:
        positions - список позицій входжень P у T

    1. якщо m = 0 або m > n:
    2.     повернути порожній список
    3.
    4. // Створюємо об'єднаний рядок
    5. S ← P + "$" + T
    6. Z ← ZAlgorithm(S)
    7. positions ← порожній список
    8.
    9. // Шукаємо позиції, де Z[i] = m
    10. для i від m + 1 до довжина(S) - 1:
    11.     якщо Z[i] = m:
    12.         // Позиція в тексті T
    13.         pos ← i - m - 1
    14.         додати pos до positions

    15. повернути positions
```

Модифікації та оптимізації

1. Z-алгоритм для циклічних рядків

Для знаходження всіх циклічних зсувів рядка, які збігаються з оригіналом:

```
Алгоритм CircularMatches(S):
    1. n ← довжина(S)
    2. doubled ← S + S  // Подвоюємо рядок
    3. Z ← ZAlgorithm(doubled)
    4. matches ← порожній список
    5.
    6. для i від 1 до n-1:
    7.     якщо Z[i] ≥ n:
    8.         додати i до matches
    9.
    10. повернути matches
```

2. Знаходження найдовшого спільного префікса

```
Алгоритм LongestCommonPrefix(S1, S2):
    1. S ← S1 + "$" + S2
    2. Z ← ZAlgorithm(S)
    3. maxLCP ← 0
    4.
    5. для i від довжина(S1) + 1 до довжина(S) - 1:
    6.     lcp ← min(Z[i], довжина(S1) - (i - довжина(S1) - 1))
    7.     maxLCP ← max(maxLCP, lcp)
    8.
    9. повернути maxLCP
```

Зв’язок з іншими алгоритмами

Z-алгоритм тісно пов’язаний з алгоритмом КМП. Фактично, префікс-функцію π алгоритму КМП можна обчислити за допомогою Z-масиву:

```
Алгоритм ConvertZtoKMP(Z):
    1. n ← довжина(Z)
    2. π ← новий масив[n]
    3. π[0] ← 0
    4.
    5. для i від 1 до n-1:
    6.     якщо Z[i] = n - i:
    7.         π[n-1] ← n - i
    8.
    9. для i від n-2 до 0 з кроком -1:
    10.    якщо π[i+1] = 0:
    11.        π[i] ← 0
    12.    інакше:
    13.        π[i] ← π[i+1] - 1
    14.
    15. для i від 1 до n-1:
    16.    якщо Z[i] > 0 і i + Z[i] - 1 < n-1:
```

```
            π[i + Z[i] - 1] ← max(π[i + Z[i] - 1], Z[i])

повернути π
```

Аналіз складності

Часова складність

Ключове спостереження для доведення лінійної складності: кожен символ рядка порівнюється з префіксом не більше двох разів:

- Один раз при розширенні Z-блоку вправо

- Один раз при використанні раніше обчисленої інформації

Отже, загальна часова складність: O(n)

Просторова складність

Алгоритм потребує O(n) додаткової пам'яті для зберігання Z-масиву.

Область застосування

Z-алгоритм широко використовується в різних галузях:

- Пошук підрядків - ефективна альтернатива КМП та Боєру-Муру

- Обчислення найдовших спільних префіксів - важливо для суфіксних масивів

- Знаходження періодичності рядків - виявлення повторюваних патернів

- Стиснення даних - алгоритми типу LZ77 використовують схожі ідеї

- Біоінформатика - аналіз повторів у ДНК послідовностях

- Обробка природної мови - пошук повторюваних фраз

- Криптоаналіз - пошук патернів у зашифрованих текстах

- Музичний аналіз - виявлення повторюваних мотивів

Приклади застосування

Приклад 1: Пошук тандемних повторів

Тандемний повтор - це підрядок виду SS, де S повторюється двічі підряд:

```
Текст: "abcabcxyzxyz"

Тандемні повтори: "abcabc" (позиція 0), "xyzxyz" (позиція 6)
```

Приклад 2: Знаходження найдовшого паліндрома

Використовуючи Z-алгоритм для оберненого рядка, можна ефективно знаходити паліндроми.

Приклад 3: Виявлення плагіату

Порівняння фрагментів тексту для знаходження спільних підрядків між документами.

Переваги та недоліки

Переваги

- Лінійна часова складність - O(n) для всіх випадків
- Простота реалізації - алгоритм концептуально простіший за КМП
- Універсальність - може бути адаптований для різних задач
- Ефективність на практиці - хороша локальність пам’яті
- Відсутність попередньої обробки - для базового варіанту
- Інтуїтивність - Z-масив має просту інтерпретацію

Недоліки

- Додаткова пам’ять - потребує O(n) пам’яті для Z-масиву
- Необхідність конкатенації - для пошуку підрядка потрібно об’єднувати рядки
- Обмеження на алфавіт - потрібен спеціальний роздільник
- Менша популярність - менше документації порівняно з КМП
- Неоптимальність для потоків - важко адаптувати для онлайн-обробки

Контрольні запитання

1. Що таке Z-масив і як він визначається для кожної позиції рядка?
2. Поясніть поняття Z-блоку та його роль в алгоритмі.
3. Чому Z-алгоритм має лінійну часову складність?
4. Як використати Z-алгоритм для пошуку всіх входжень підрядка?
5. В чому різниця між випадками $i > r$ та $i \leqslant r$ в основному циклі?
6. Як пов’язані Z-масив та префікс-функція алгоритму КМП?
7. Чому потрібен спеціальний символ-роздільник при пошуку підрядка?
8. Які задачі, крім пошуку підрядка, можна розв’язати за допомогою Z-алгоритму?
9. Як модифікувати алгоритм для знаходження найдовшого спільного підрядка?
10. В яких випадках Z-алгоритм ефективніший за інші алгоритми пошуку?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження Z-алгоритму для обробки рядків

2. Мета

Вивчити принципи роботи Z-алгоритму, реалізувати його для обчислення Z-масиву та пошуку підрядків, дослідити застосування для різних задач обробки рядків.

3. Завдання

- Реалізувати базовий Z-алгоритм для обчислення Z-масиву
- Реалізувати функцію пошуку підрядка за допомогою Z-алгоритму
- Створити візуалізацію процесу обчислення Z-масиву
- Реалізувати застосування для знаходження тандемних повторів
- Порівняти продуктивність з алгоритмами КМП та наївним пошуком
- Дослідити поведінку алгоритму на різних типах даних

4. Опис

Лабораторна робота передбачає глибоке вивчення Z-алгоритму через практичну реалізацію та дослідження. Студенти мають не лише реалізувати базовий алгоритм, але й дослідити його застосування для різних задач, створити візуалізацію для кращого розуміння роботи алгоритму та провести порівняльний аналіз з іншими алгоритмами пошуку підрядків.

5. Порядок виконання

- Створити клас ZAlgorithm з методом computeZArray
- Реалізувати покрокову версію з виведенням проміжних станів
- Імплементувати метод searchPattern для пошуку підрядка
- Додати методи для знаходження тандемних повторів
- Створити візуалізацію Z-блоків та процесу обчислення
- Реалізувати метод findAllPeriods для знаходження періодів
- Провести тестування на різних типах рядків
- Виконати порівняльний аналіз продуктивності

6. Результат

Програма повинна: - Коректно обчислювати Z-масив для довільних рядків - Знаходити всі входження підрядка в тексті - Візуалізувати процес роботи алгоритму - Виявляти періодичні структури в рядках - Демонструвати ефективність на великих даних - Порівнювати результати з іншими алгоритмами

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас ZAlgorithm з основними методами

- Метод computeZArray з детальними коментарями

- Метод searchPattern для пошуку підрядків

- Методи для спеціальних застосувань (тандемні повтори, періоди)

- Модуль візуалізації процесу обчислення

- Модуль тестування та бенчмаркінгу

2. Продемонструвати роботу додатку:

- Покрокове обчислення Z-масиву з візуалізацією

- Пошук підрядків у різних текстах

- Знаходження тандемних повторів

- Аналіз періодичності рядків

- Порівняння часу виконання з іншими алгоритмами

- Робота з великими файлами

3. Бути готовим відповісти на питання:

- Як обчислюється Z[i] для конкретних прикладів?

- Чому алгоритм має лінійну складність?

- Як вибрати оптимальний роздільник для пошуку?

- Як модифікувати алгоритм для специфічних задач?

- В чому переваги Z-алгоритму над КМП?

Тести

1. Z[i] для рядка визначається як:

1) Довжина найдовшого суфікса, що збігається з префіксом

2) Довжина найдовшого префікса, що починається з позиції i

3) Кількість входжень символу S[i] у префіксі

4) Відстань до найближчого повтору

2. Часова складність Z-алгоритму становить:

1) $O(n^2)$

2) $O(n \log n)$

3) $O(n)$

4) $O(n \times m)$

3. Для пошуку підрядка P у тексті T створюється рядок:

1) T + P

2) P + T

3) P + “$” + T

4) T + “$” + P

4. Z-блок [l, r] означає, що:

1) S[l..r] = S[0..r-l]

2) S[l..r] = S[r..l]

3) S[0..l] = S[0..r]

4) S[l] = S[r]

5. При $i \leqslant r$ в алгоритмі використовується значення:

1) Z[i]

2) Z[r]

3) Z[i-l]

4) Z[r-l]

Відповіді:

1. b)

2. c)

3. c)

4. a)

5. c)

Тести підвищеної складності

1. Для рядка “abacaba” значення Z[3] дорівнює:

1) 0

2) 1

3) 3

4) 4

2. Максимально можливе значення Z[i] для i > 0:

1) n

2) n - 1

3) n - i

4) i

3. Скільки разів кожен символ порівнюється в Z-алгоритмі?

1) Рівно один раз

2) Не більше двох разів

3) O(log n) разів

4) Залежить від рядка

4. Для знаходження всіх періодів рядка довжини n потрібно перевірити Z[i] для:

1) i від 1 до n-1

2) i від 1 до n/2

3) i кратних періоду

4) i = n - період

5. При обчисленні Z[i], якщо Z[k] = r - i + 1, то:

1) Z[i] = Z[k]

2) Z[i] ⩾ Z[k]

3) Z[i] ⩽ Z[k]

4) Потрібна додаткова перевірка

6. Для рядка з n однакових символів, скільки значень Z[i] = n - i?

1) 0

2) 1

3) n - 1

4) n

7. Z-алгоритм можна використати для обчислення префікс-функції КМП за час:

1) $O(n^2)$

2) $O(n \log n)$

3) $O(n)$

4) Неможливо

8. Для виявлення всіх тандемних повторів потрібно перевірити:

1) $Z[i] \geqslant i$ для всіх i

2) $Z[i] = i$ для деяких i

3) $Z[i] \geqslant$ довжина повтору

4) Спеціальну умову для кожної позиції

9. При модифікації для циклічних рядків використовується:

1) Z-масив подвоєного рядка

2) Два окремі Z-масиви

3) Модифікована функція порівняння

4) Інший алгоритм

10. Простір, необхідний для зберігання всієї інформації Z-алгоритму:

1) $O(1)$

2) $O(|\Sigma|)$, де $|\Sigma|$ - розмір алфавіту

3) $O(n)$

4) $O(n \log n)$

Відповіді:

1. a) S[3] = ‘c’ не збігається з S[0] = ‘a’ 2.

c) максимальна можлива довжина підрядка з позиції i

3. b) Кожен символ порівнюється максимум двічі завдяки Z-блокам

4. a) Період p існує, якщо $Z[n-p] \geqslant p$

5. d) Потрібно перевірити символи за межами поточного Z-блоку

6. c) Всі позиції крім першої мають максимальні значення

7. c) Існує лінійний алгоритм конвертації

8. a) Тандемний повтор на позиції i існує, якщо $Z[i] \geqslant i$

9. а) Використовується подвоєний рядок S + S

10. с) Потрібен лише Z-масив розміру n

Завдання для самостійної роботи

1. Реалізуйте модифікацію Z-алгоритму для пошуку найдовшого паліндромного підрядка, використовуючи обернений рядок.

2. Створіть алгоритм для знаходження всіх примітивних (неперіодичних) підрядків заданої довжини за допомогою Z-масиву.

3. Розробіть ефективний алгоритм для обчислення кількості різних підрядків у рядку, використовуючи Z-алгоритм.

4. Імплементуйте алгоритм для знаходження найменшого періоду рядка та всіх його кратних періодів.

5. Створіть модифікацію для приблизного пошуку підрядків з не більше ніж k невідповідностями.

6. Реалізуйте паралельну версію Z-алгоритму для обробки великих рядків на багатоядерних системах.

7. Розробіть алгоритм для знаходження всіх квазіперіодичних структур у рядку (періоди з невеликими варіаціями).

8. Створіть адаптивну версію, яка вибирає між Z-алгоритмом та іншими алгоритмами залежно від характеристик даних.

9. Імплементуйте двовимірний Z-алгоритм для обробки матриць символів.

10. Розробіть застосування Z-алгоритму для стиснення даних методом виявлення повторюваних фрагментів.

Теми лабораторних робіт

1. Розробка системи виявлення плагіату в програмному коді з використанням Z-алгоритму для пошуку схожих фрагментів.

2. Створення біоінформатичного інструменту для аналізу тандемних повторів та періодичних структур у ДНК.

3. Імплементація музичного аналізатора для виявлення повторюваних мотивів та структур у MIDI-файлах.

4. Розробка системи стиснення тексту на основі виявлення та кодування повторюваних патернів.

5. Створення інструменту для аналізу логів веб-сервера з виявленням періодичних патернів активності.

6. Реалізація системи пошуку дублікатів у великих текстових корпусах з оптимізацією для SSD.

7. Розробка криптоаналітичного інструменту для пошуку слабких місць у псевдовипадкових послідовностях.

8. Створення редактора з інтелектуальним автодоповненням на основі аналізу повторюваних патернів.

9. Імплементація системи виявлення аномалій у часових рядах через аналіз порушень періодичності.

10. Розробка інструменту для оптимізації регулярних виразів шляхом виявлення спільних підпатернів.

10.3. Редакційна відстань

У сучасному світі обробки текстової інформації часто виникає потреба визначити, наскільки схожими або відмінними є два рядки. Чи то виправлення орфографічних помилок у текстовому редакторі, чи пошук найближчих відповідників у базі даних, чи аналіз генетичних послідовностей у біоінформатиці – всі ці задачі потребують формального способу вимірювання “відстані” між рядками.

Редакційна відстань– це міра різниці між двома послідовностями символів, яка визначається як мінімальна кількість операцій редагування, необхідних для перетворення одного рядка в інший. Ця концепція стала фундаментальною в теорії обробки рядків та знайшла широке застосування в різноманітних галузях – від комп’ютерної лінгвістики до молекулярної біології.

Розділ присвячено трьом ключовим алгоритмам обчислення редакційної відстані, кожен з яких має свої особливості та сфери застосування:

Відстань Левенштейна – класична метрика, яка враховує три базові операції редагування: вставку, видалення та заміну символів. Запропонована Володимиром Левенштейном у 1965 році, вона стала основою для багатьох подальших розробок у цій галузі.

Відстань Дамерау-Левенштейна – розширення класичної відстані Левенштейна, яке додатково враховує операцію транспозиції (перестановки двох сусідніх символів). Ця модифікація особливо корисна для виявлення типових помилок при введенні тексту, коли користувач випадково міняє місцями сусідні літери.

Алгоритм Гіршберга – елегантне рішення проблеми високої просторової складності базових алгоритмів. Використовуючи техніку “розділяй і володарюй”, алгоритм дозволяє обчислити не лише саму відстань, але й оптимальну послідовність операцій редагування, використовуючи лише лінійний обсяг пам’яті.

У цьому розділі ми детально розглянемо математичні основи кожного алгоритму, їх програмну реалізацію, аналіз складності та практичні застосування. Особливу увагу буде приділено порівнянню ефективності різних підходів та вибору оптимального алгоритму для конкретних задач. Студенти навчаться не лише використовувати готові реалізації, але й глибоко розуміти принципи роботи цих алгоритмів, що дозволить модифікувати їх для специфічних потреб та створювати власні варіації для нових застосувань.

10.2.1. Алгоритм обчислення відстані Левенштейна

Теоретичні відомості

Відстань Левенштейна, також відома як редакційна відстань, є мірою різниці між двома послідовностями символів. Вона визначається як мінімальна кількість односимвольних операцій редагування, необхідних для перетворення одного рядка в інший. Алгоритм був розроблений Володимиром Левенштейном у 1965 році для вирішення задач теорії кодування та виправлення помилок.

Формально, для двох рядків s_1 та s_2, відстань Левенштейна $d(s_1, s_2)$ - це мінімальна кількість операцій з наступного набору:

- **Вставка** символу в рядок

- **Видалення** символу з рядка

- **Заміна** одного символу на інший

Основою алгоритму є матриця відстаней. Розглянемо її побудову.

Побудова матриці відстаней (DP Matrix)

Створимо матрицю `D` розміром `(m+1) x (n+1)`, де `m` — довжина першого рядка `S1`, а `n` — довжина другого рядка `S2`.

Кожна комірка `D[i][j]` в цій матриці буде зберігати відстань Левенштейна між першими `i` символами рядка `S1` (тобто `S1[0...i-1]`) та першими `j` символами рядка `S2` (тобто `S2[0...j-1]`).

Крок 1: Ініціалізація

- Перший рядок: `D[0][j]` представляє вартість перетворення порожнього рядка в префікс `S2` довжиною `j`. Це вимагає `j` операцій вставки. Отже, `D[0][j] = j`.

- Перший стовпець: `D[i][0]` представляє вартість перетворення префікса `S1` довжиною `i` в порожній рядок. Це вимагає `i` операцій видалення. Отже, `D[i][0] = i`.

Крок 2: Обчислення решти комірок (рекурентне співвідношення)

Ми заповнюємо матрицю зліва направо, зверху вниз. Щоб обчислити значення `D[i][j]`, ми розглядаємо три можливі операції, які призводять нас до цього стану:

1. **Видалення:** Ми можемо перетворити `S1[0...i-1]` в `S2[0...j-1]`, якщо ми вже вміємо перетворювати `S1[0...i-2]` в `S2[0...j-1]` (вартість цього `D[i-1][j]`) і просто видалимо останній символ `S1[i-1]`.

Вартість: `D[i-1][j] + 1`.

2. **Вставка:** Ми можемо досягти того ж результату, якщо ми вже вміємо перетворювати `S1[0...i-1]` в `S2[0...j-2]` (вартість `D[i][j-1]`) і просто вставимо останній символ `S2[j-1]`.

Вартість: `D[i][j-1] + 1`.

3. **Заміна/Збіг:** Ми можемо перетворити `S1[0...i-1]` в `S2[0...j-1]`, виходячи зі стану `D[i-1][j-1]` (перетворення `S1[0...i-2]` в `S2[0...j-2]`). Далі ми дивимося на останні символи: `S1[i-1]` та `S2[j-1]`.

- Якщо `S1[i-1] == S2[j-1]`, символи збігаються, і додаткових операцій не потрібно.

Вартість: `D[i-1][j-1] + 0`.

- Якщо `S1[i-1] != S2[j-1]`, потрібна заміна.

Вартість: `D[i-1][j-1] + 1`.

Значення `D[i][j]` є мінімумом з цих трьох варіантів.

$$D[i][j] = \min\{D[i-1][j] + 1 \quad \text{(видалення)}\ D[i][j-1] + 1 \quad \text{(вставка)}\ D[i-1][j-1] + \text{cost}$$

де `cost = 0`, якщо `S1[i-1] == S2[j-1]`, і `cost = 1`, якщо вони різні.

Крок 3: Результат Фінальна відстань Левенштейна між рядками `S1` та `S2` знаходиться в правій нижній комірці матриці: `D[m][n]`.

Розглянемо побудову на прикладі.

`S1 = "BITING"`, `S2 = "SITTING"`

`m=6`, `n=7`. Створюємо матрицю 7x8.

1. Ініціалізація:

		S	I	T	T	I	N	G
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
B	**1**							
I	**2**							
T	**3**							
I	**4**							
N	**5**							
G	**6**							

2. Обчислення `D[1][1]` (`B` -> `S`):

- `D[0][1]+1` (вставка S) = 1+1=2
- `D[1][0]+1` (видалення B) = 1+1=2
- `D[0][0]+1` (заміна B->S) = 0+1=1
- `D[1][1] = min(2,2,1) = 1`

3. Обчислення `D[2][2]` (`BI` -> `SI`): `S1[1]`='I', `S2[1]`='I'. Символи однакові.

- `D[1][2]+1` (вставка I) = 2+1=3
- `D[2][1]+1` (видалення I) = 2+1=3
- `D[1][1]+0` (збіг I==I) = 1+0=1
- `D[2][2] = min(3,3,1) = 1`

Фінальна матриця:

		S	I	T	T	I	N	G
	0	1	2	3	4	5	6	7
B	1	1	2	3	4	5	6	7
I	2	2	1	2	3	4	5	6
T	3	3	2	1	2	3	4	5
I	4	4	3	2	2	2	3	4

		S	I	T	T	I	N	G
N	5	5	4	3	3	3	2	3
G	6	6	5	4	4	4	3	2

Результат: **D[6][7] = 2**. Відстань між “BITING” та “SITTING” дорівнює 2. (Наприклад: **BITING** -> **SITING** (заміна B->S) -> **SITTING** (вставка T)).

Випишемо остаточне рекурентне співвідношення для цієї матриці.

Нехай маємо два рядки: - s_1 довжиною m з символами $s_1[1]$, $s_1[2]$, …, $s_1[m]$ - s_2 довжиною n з символами $s_2[1]$, $s_2[2]$, …, $s_2[n]$

Визначимо D[i,j] як відстань Левенштейна між префіксами $s_1[1..i]$ та $s_2[1..j]$. Тоді рекурентне співвідношення має вигляд:

```
D[0,0] = 0
D[i,0] = i для всіх i від 1 до m
D[0,j] = j для всіх j від 1 до n

D[i,j] = min {
    D[i-1,j] + 1,                   // видалення
    D[i,j-1] + 1,                   // вставка
    D[i-1,j-1] + cost(i,j)          // заміна або збіг
}

де cost(i,j) = 0, якщо s₁[i] = s₂[j]
               1, якщо s₁[i] ≠ s₂[j]
```

Основний алгоритм використовує метод динамічного програмування для ефективного обчислення відстані:

```
Алгоритм LevenshteinDistance(s1, s2):
    Вхід:
        s1 - перший рядок довжиною m
        s2 - другий рядок довжиною n
    Вихід:
        distance - відстань Левенштейна між s1 та s2

    m ← довжина(s1)
    n ← довжина(s2)

    // Створюємо матрицю для динамічного програмування
    D ← новий масив[0..m, 0..n]

    // Базові випадки: перетворення з/в порожній рядок
    для i від 0 до m:
        D[i,0] ← i
```

```
    для j від 0 до n:
        D[0,j] ← j

    // Заповнюємо матрицю
    для i від 1 до m:
        для j від 1 до n:
            // Вартість заміни: 0 якщо символи однакові, 1 якщо різні
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← 1

            // Обираємо мінімум з трьох операцій
            D[i,j] ← min(
                D[i-1,j] + 1,      // видалення з s1
                D[i,j-1] + 1,      // вставка в s1
                D[i-1,j-1] + cost  // заміна або збіг
            )

    повернути D[m,n]
```

Розглянемо детально, як працює алгоритм на прикладі перетворення слова “SATURDAY” у “SUNDAY”.

Крок 1: Ініціалізація Створюємо матрицю розміром (m+1) × (n+1), де m = 8 (довжина “SATURDAY”) і n = 6 (довжина “SUNDAY”).

Крок 2: Заповнення базових випадків

- Перший рядок: відстань від порожнього рядка до префіксів “SUNDAY”

- Перший стовпець: відстань від префіксів “SATURDAY” до порожнього рядка

Крок 3: Заповнення матриці Для кожної клітинки D[i,j] обчислюємо мінімум з трьох можливих шляхів:

- Прийти з D[i-1,j] + 1 (видалити символ з першого рядка)

- Прийти з D[i,j-1] + 1 (вставити символ у перший рядок)

- Прийти з D[i-1,j-1] + cost (замінити символ або залишити без змін)

Детальний процес заповнення для позиції D[1,1]:

- s1[1] = ‘S’, s2[1] = ‘S’ → символи однакові, cost = 0

- D[0,1] + 1 = 1 + 1 = 2 (видалення)

- D[1,0] + 1 = 1 + 1 = 2 (вставка)

- D[0,0] + 0 = 0 + 0 = 0 (збіг)

- D[1,1] = min(2, 2, 0) = 0

Для отримання не лише відстані, але й самої послідовності операцій, використовується зворотний прохід по матриці:

```
Алгоритм GetEditOperations(D, s1, s2):
    Вхід:
        D - заповнена матриця відстаней
        s1, s2 - вхідні рядки
    Вихід:
        operations - список операцій редагування

    operations ← порожній список
    i ← довжина(s1)
    j ← довжина(s2)

    поки i > 0 або j > 0:
        якщо i = 0:
            додати "Вставити s2[j]" до operations
            j ← j - 1
        інакше якщо j = 0:
            додати "Видалити s1[i]" до operations
            i ← i - 1
        інакше:
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← 1

            // Визначаємо, з якої клітинки прийшли
            якщо D[i,j] = D[i-1,j-1] + cost:
                якщо cost = 0:
                    додати "Залишити s1[i]" до operations
                інакше:
                    додати "Замінити s1[i] на s2[j]" до operations
                i ← i - 1
                j ← j - 1
            інакше якщо D[i,j] = D[i-1,j] + 1:
                додати "Видалити s1[i]" до operations
                i ← i - 1
            інакше:
                додати "Вставити s2[j]" до operations
                j ← j - 1
```

```
    розвернути operations
    повернути operations
```

Розглянемо покроковий процес обчислення відстані між словами “CAT” та “DOG”:

Ініціалізація матриці:

```
    ""  D  O  G
""   0  1  2  3
C    1
A    2
T    3
```

Заповнення першого рядка (i=1, “C”):

- D[1,1]: min(D[0,1]+1=2, D[1,0]+1=2, D[0,0]+1=1) = 1 (C≠D, заміна)

- D[1,2]: min(D[0,2]+1=3, D[1,1]+1=2, D[0,1]+1=2) = 2

- D[1,3]: min(D[0,3]+1=4, D[1,2]+1=3, D[0,2]+1=3) = 3

Заповнення другого рядка (i=2, “CA”):

- D[2,1]: min(D[1,1]+1=2, D[2,0]+1=3, D[1,0]+1=2) = 2

- D[2,2]: min(D[1,2]+1=3, D[2,1]+1=3, D[1,1]+1=2) = 2

- D[2,3]: min(D[1,3]+1=4, D[2,2]+1=3, D[1,2]+1=3) = 3

Заповнення третього рядка (i=3, “CAT”):

- D[3,1]: min(D[2,1]+1=3, D[3,0]+1=4, D[2,0]+1=3) = 3

- D[3,2]: min(D[2,2]+1=3, D[3,1]+1=4, D[2,1]+1=3) = 3

- D[3,3]: min(D[2,3]+1=4, D[3,2]+1=4, D[2,2]+1=3) = 3

Фінальна матриця:

```
    ""  D  O  G
""   0  1  2  3
C    1  1  2  3
A    2  2  2  3
T    3  3  3  3
```

Відстань Левенштейна між “CAT” та “DOG” дорівнює 3.

Якщо потрібна лише відстань (без послідовності операцій), можна оптимізувати використання пам’яті до O(min(m,n)):

```
Алгоритм LevenshteinDistanceOptimized(s1, s2):
    Вхід:
        s1, s2 - вхідні рядки
    Вихід:
        distance - відстань Левенштейна
```

```
    // Забезпечуємо, що s1 - коротший рядок
    якщо довжина(s1) > довжина(s2):
        поміняти місцями s1 та s2

    m ← довжина(s1)
    n ← довжина(s2)

    // Використовуємо лише два рядки замість повної матриці
    prevRow ← новий масив[0..m]
    currRow ← новий масив[0..m]

    // Ініціалізація першого рядка
    для i від 0 до m:
        prevRow[i] ← i

    для j від 1 до n:
        currRow[0] ← j

        для i від 1 до m:
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← 1

            currRow[i] ← min(
                prevRow[i] + 1,      // видалення
                currRow[i-1] + 1,    // вставка
                prevRow[i-1] + cost  // заміна
            )

        // Міняємо рядки місцями
        temp ← prevRow
        prevRow ← currRow
        currRow ← temp

    повернути prevRow[m]
```

Розглянемо модифікації алгоритму.

1. Зважена відстань Левенштейна

У деяких застосуваннях різні операції можуть мати різну вартість:

```
Алгоритм WeightedLevenshteinDistance(s1, s2, insertCost, deleteCost,
replaceCost):
    m ← довжина(s1)
```

```
    n ← довжина(s2)
    D ← новий масив[0..m, 0..n]

    // Ініціалізація з врахуванням ваг
    D[0,0] ← 0
    для i від 1 до m:
        D[i,0] ← D[i-1,0] + deleteCost
    для j від 1 до n:
        D[0,j] ← D[0,j-1] + insertCost

    для i від 1 до m:
        для j від 1 до n:
            якщо s1[i-1] = s2[j-1]:
                matchCost ← 0
            інакше:
                matchCost ← replaceCost

            D[i,j] ← min(
                D[i-1,j] + deleteCost,
                D[i,j-1] + insertCost,
                D[i-1,j-1] + matchCost
            )

    повернути D[m,n]
```

2. Обмежена відстань Левенштейна

Для прискорення обчислень при пошуку схожих рядків можна обмежити максимальну відстань:

```
Алгоритм BoundedLevenshteinDistance(s1, s2, maxDistance):
    m ← довжина(s1)
    n ← довжина(s2)

    // Швидка перевірка
    якщо |m - n| > maxDistance:
        повернути maxDistance + 1

    // Створюємо смугу навколо діагоналі
    D ← новий масив[0..m, 0..n]

    для i від 0 до m:
        для j від max(0, i-maxDistance) до min(n, i+maxDistance+1):
            якщо i = 0:
                D[i,j] ← j
            інакше якщо j = 0:
                D[i,j] ← i
```

```
            інакше:
                якщо s1[i-1] = s2[j-1]:
                    cost ← 0
                інакше:
                    cost ← 1

                D[i,j] ← min(
                    D[i-1,j] + 1 якщо j <= i+maxDistance-1,
                    D[i,j-1] + 1 якщо j >= i-maxDistance+1,
                    D[i-1,j-1] + cost
                )

    якщо D[m,n] <= maxDistance:
        повернути D[m,n]
    інакше:
        повернути maxDistance + 1
```

Для певних цілей корисно розглянути рекурсивну версію:

```
Алгоритм LevenshteinRecursive(s1, s2, memo):
    m ← довжина(s1)
    n ← довжина(s2)

    // Базові випадки
    якщо m = 0:
        повернути n
    якщо n = 0:
        повернути m

    // Перевірка мемоізації
    якщо (m, n) в memo:
        повернути memo[(m, n)]

    // Рекурсивні виклики
    якщо s1[m-1] = s2[n-1]:
        result ← LevenshteinRecursive(s1[0..m-1], s2[0..n-1], memo)
    інакше:
        delete ← LevenshteinRecursive(s1[0..m-1], s2, memo) + 1
        insert ← LevenshteinRecursive(s1, s2[0..n-1], memo) + 1
        replace ← LevenshteinRecursive(s1[0..m-1], s2[0..n-1], memo)
+ 1
        result ← min(delete, insert, replace)

    memo[(m, n)] ← result
    повернути result
```

Аналіз складності

Часова складність:

- Базовий алгоритм: $O(m \times n)$

- З обмеженням відстані k: $O(k \times min(m,n))$

- Рекурсивний з мемоізацією: $O(m \times n)$

Просторова складність:

- Повна матриця: $O(m \times n)$

- Оптимізована версія: $O(min(m,n))$

- Лише для відстані з обмеженням k: $O(k)$

- Рекурсивна з мемоізацією: $O(m \times n)$

Властивості відстані Левенштейна

Відстань Левенштейна є метрикою, тобто задовольняє такі властивості:

1. Невід'ємність: $d(x, y) \geqslant 0$ для всіх x, y

2. Тотожність: $d(x, y) = 0$ тоді і тільки тоді, коли $x = y$

3. Симетричність: $d(x, y) = d(y, x)$

4. Нерівність трикутника: $d(x, z) \leqslant d(x, y) + d(y, z)$

Ці властивості роблять відстань Левенштейна корисною для багатьох застосувань, включаючи кластеризацію та пошук найближчих сусідів.

Область застосування

Алгоритм Левенштейна широко використовується в різних галузях:

- Обробка природної мови - виправлення орфографічних помилок, пошук схожих слів

- Біоінформатика - порівняння генетичних послідовностей, виявлення мутацій

- Розпізнавання мовлення - корекція результатів розпізнавання

- Системи рекомендацій - пошук схожих елементів за назвами

- Дедуплікація даних - виявлення майже ідентичних записів

- Контроль версій - обчислення різниці між версіями файлів

- Пошукові системи - виправлення запитів користувачів

- Машинний переклад - оцінка якості перекладу

Приклади застосування

Приклад 1: Виправлення орфографічних помилок

```
Користувач ввів: "алгоритм"
Словник: ["алгоритм", "алгебра", "алгоритм", "логарифм"]

Відстані Левенштейна:
- "алгоритм" → "алгоритм": 0
- "алгоритм" → "алгебра": 5
- "алгоритм" → "логарифм": 4

Пропонується виправлення: "алгоритм"
```

Приклад 2: Порівняння ДНК послідовностей

```
Послідовність 1: AGCTGACG
Послідовність 2: AGCTTACG

Відстань Левенштейна: 1 (одна вставка 'T')
Це може вказувати на точкову мутацію
```

Приклад 3: Пошук дублікатів у базі даних

```
Запис 1: "Іванов Іван Іванович"
Запис 2: "Іванов І.І."
Запис 3: "Іванов Іван"

Використовуючи нормалізовану відстань Левенштейна,
можна виявити потенційні дублікати
```

Переваги та недоліки

Переваги

- Простота реалізації - алгоритм інтуїтивно зрозумілий та легко програмується

- Універсальність - працює з будь-якими послідовностями символів

- Точність - дає оптимальну (мінімальну) кількість операцій

- Гнучкість - легко модифікується для різних вартостей операцій

- Математична обґрунтованість - є метрикою в математичному сенсі

- Широка підтримка - реалізований у багатьох бібліотеках

Недоліки

- Висока обчислювальна складність - O(m×n) для довгих рядків

- Велике використання пам’яті - O(m×n) для збереження матриці

- Не враховує транспозиції - перестановка сусідніх символів рахується як дві операції

- Відсутність контексту - всі символи вважаються однаково важливими

- Погано масштабується - неефективний для дуже довгих послідовностей

- Не враховує семантику - для природної мови може давати неінтуїтивні результати

Контрольні запитання

1. Що таке відстань Левенштейна і які операції вона враховує?

2. Яка рекурентна формула використовується для обчислення відстані?

3. Чому алгоритм використовує динамічне програмування?

4. Як відновити послідовність операцій редагування?

5. Яка часова та просторова складність базового алгоритму?

6. Як можна оптимізувати використання пам’яті?

7. Чим відрізняється зважена відстань Левенштейна?

8. Для яких задач доцільно використовувати обмежену відстань?

9. Чи є відстань Левенштейна метрикою? Обґрунтуйте.

10. Які існують альтернативи відстані Левенштейна?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму обчислення відстані Левенштейна

2. Мета

Вивчити принципи динамічного програмування на прикладі алгоритму Левенштейна, реалізувати різні варіації алгоритму, дослідити його ефективність та застосування для практичних задач.

3. Завдання

- Реалізувати базовий алгоритм обчислення відстані Левенштейна

- Реалізувати функцію відновлення послідовності операцій

- Створити оптимізовану версію з O(min(m,n)) пам’яті

- Реалізувати зважену версію алгоритму

- Розробити візуалізацію процесу заповнення матриці

- Реалізувати пошук найближчих слів у словнику

- Провести аналіз продуктивності для різних розмірів вхідних даних

4. Опис

Лабораторна робота передбачає поглиблене вивчення алгоритму Левенштейна через практичну реалізацію його різних варіацій. Студенти мають не лише реалізувати базовий алгоритм, але й дослідити його модифікації, створити інструменти візуалізації та розробити практичні застосування для реальних задач.

5. Порядок виконання

- Створити клас LevenshteinCalculator з основними методами
- Реалізувати метод calculateDistance для обчислення відстані
- Імплементувати метод getEditOperations для відновлення операцій
- Додати метод calculateDistanceOptimized з оптимізованою пам’яттю
- Створити метод weightedDistance з настроюваними вагами
- Реалізувати візуалізатор MatrixVisualizer
- Розробити клас SpellChecker на основі алгоритму
- Провести тестування на різних типах даних
- Виконати порівняльний аналіз різних реалізацій

6. Результат

Готова система повинна: - Коректно обчислювати відстань Левенштейна між рядками - Відновлювати оптимальну послідовність операцій редагування - Візуалізувати процес динамічного програмування - Ефективно працювати з великими рядками - Демонструвати практичні застосування - Надавати статистику продуктивності

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас LevenshteinCalculator з усіма методами обчислення
- Метод calculateDistance з повною матрицею
- Метод getEditOperations для відновлення шляху
- Метод calculateDistanceOptimized з оптимізацією пам’яті
- Клас SpellChecker для виправлення помилок
- Модуль візуалізації процесу обчислення
- Модуль тестування та бенчмаркінгу

2. Продемонструвати роботу додатку:

- Обчислення відстані для різних пар рядків
- Візуалізація заповнення матриці покроково
- Відновлення та демонстрація послідовності операцій
- Порівняння використання пам’яті різних версій
- Робота системи виправлення орфографічних помилок
- Графіки залежності часу виконання від розміру вхідних даних

3. Бути готовим відповісти на питання:
- Як заповнюється матриця для конкретної пари рядків?
- Чому обирається саме мінімум з трьох значень?
- Як працює відновлення шляху редагування?
- В чому переваги оптимізованої версії?
- Як вибрати оптимальні ваги для конкретної задачі?

Тести

1. Відстань Левенштейна між рядками “kitten” та “sitting” дорівнює:

1) 2

2) 3

3) 4

4) 5

2. Які операції враховує класична відстань Левенштейна?

1) Лише заміну символів

2) Вставку та видалення

3) Вставку, видалення та заміну

4) Вставку, видалення, заміну та транспозицію

3. Часова складність базового алгоритму Левенштейна:

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) O(m × n)

4. Мінімальна просторова складність для обчислення лише відстані:

1) O(1)

2) O(min(m, n))

3) O(m + n)

4) O(m × n)

5. Відстань Левенштейна є метрикою, тому що вона задовольняє:

1) Лише невід’ємність

2) Симетричність та нерівність трикутника

3) Всі аксіоми метрики

4) Лише транзитивність

Відповіді:

1. b)

2. c)

3. d)

4. b)

5. c)

Тести підвищеної складності

1. Для рядків довжини n з алфавіту розміру σ, скільки різних матриць може згенерувати алгоритм Левенштейна?

1) σ^n

2) n!

3) Залежить від конкретних рядків

4) 2^n

2. При обчисленні відстані між рядками “algorithm” та “altruistic”, яке значення буде в клітинці D[4,4]?

1) 1

2) 2

3) 3

4) 4

3. Якщо обмежити максимальну відстань значенням k, яка буде часова складність?

1) $O(n)$

2) $O(k^2)$

3) $O(k \times \min(m,n))$

4) $O(m \times n)$

4. Для якої задачі НЕ підходить стандартна відстань Левенштейна?

1) Виправлення орфографічних помилок

2) Виявлення анаграм

3) Порівняння ДНК послідовностей

4) Пошук дублікатів

5. При використанні лише двох рядків для оптимізації пам’яті, що втрачається?

1) Точність обчислення

2) Можливість відновити операції

3) Швидкість обчислення

4) Нічого для обчислення відстані

6. Нормалізована відстань Левенштейна обчислюється як:

1) $d(s_1,s_2) / \min(|s_1|,|s_2|)$

2) $d(s_1,s_2) / \max(|s_1|,|s_2|)$

3) $d(s_1,s_2) / (|s_1| + |s_2|)$

4) $d(s_1,s_2) \times 2 / (|s_1| + |s_2|)$

7. Для паралельного обчислення відстані Левенштейна найефективніше:

1) Розділити матрицю по рядках

2) Розділити матрицю по стовпцях

3) Обчислювати по діагоналях

4) Використати рекурсію з мемоізацією

8. Якщо вартість заміни залежить від пари символів, скільки додаткової пам’яті потрібно?

1) $O(1)$

2) $O(\sigma)$

3) $O(\sigma^2)$

4) O(n)

9. Відстань Левенштейна між рядком та його реверсом:

1) Завжди дорівнює довжині рядка

2) Завжди парна

3) Може бути будь-якою

4) Завжди не більше довжини рядка

10. Для знаходження k найближчих слів у словнику розміру N найкраще:

1) Обчислити всі відстані та відсортувати

2) Використати BK-дерево

3) Використати трі-структуру

4) Застосувати хешування

Відповіді:

1. c) Залежить від конкретних рядків - матриця визначається вхідними даними

2. b) 2 - “algo” → “altr” потребує 2 операції

3. c) O(k × min(m,n)) - обчислюємо лише смугу шириною 2k+1

4. b) - Анаграми мають велику відстань Левенштейна попри однаковий набір літер

5. b) - Без повної матриці неможливо відновити послідовність операцій

6. b) $d(s_1,s_2) / \max(|s_1|,|s_2|)$ - нормалізація по довжині довшого рядка

7. c) - Елементи на діагоналі не залежать один від одного

8. c) $O(\sigma^2)$ - потрібна матриця вартостей для всіх пар символів

9. c) - Залежить від конкретного рядка, наприклад “abc”↔“cba” має відстань 2

10. b) - BK-дерево оптимізоване для метричного пошуку

Завдання для самостійної роботи

1. Реалізуйте алгоритм обчислення відстані Левенштейна для Unicode рядків з правильною обробкою багатобайтових символів та емодзі.

2. Створіть паралельну версію алгоритму, яка обчислює матрицю по діагоналях з використанням багатопоточності.

3. Розробіть адаптивний алгоритм, який автоматично вибирає між повною матрицею та оптимізованою версією залежно від доступної пам’яті.

4. Імплементуйте узагальнену відстань Левенштейна для послідовностей об’єктів довільного типу з настроюваною функцією порівняння.

5. Створіть алгоритм приблизного обчислення відстані Левенштейна з гарантованою точністю для дуже довгих рядків.

6. Реалізуйте версію з кешуванням для ефективного обчислення відстаней між рядками з спільними префіксами.

7. Розробіть модифікацію для обчислення відстані між деревоподібними структурами.

8. Створіть алгоритм для знаходження найдовшої спільної підпослідовності на основі матриці Левенштейна.

9. Імплементуйте потоковий алгоритм для обчислення відстані між рядками, що надходять посимвольно.

10. Розробіть систему автоматичного визначення оптимальних ваг операцій на основі корпусу прикладів.

Теми лабораторних робіт

1. Розробка інтелектуальної системи виправлення орфографічних помилок з врахуванням контексту та частотності слів.

2. Створення системи виявлення плагіату в програмному коді з використанням нормалізованої відстані Левенштейна.

3. Імплементація біоінформатичного інструменту для аналізу мутацій у генетичних послідовностях.

4. Розробка системи дедуплікації записів у базі даних клієнтів з нечітким пошуком.

5. Створення рекомендаційної системи для пошукових запитів на основі історії та виправлення помилок.

6. Реалізація системи порівняння та злиття версій документів з візуалізацією змін.

7. Розробка інструменту для оцінки якості машинного перекладу порівнянням з еталонними текстами.

8. Створення системи класифікації текстів за схожістю для автоматичної категоризації.

9. Імплементація музичного пошуковика, що знаходить схожі мелодії за послідовністю нот.

10. Розробка системи автоматичного зв'язування сутностей у текстах з використанням нечіткого порівняння.

10.2.2. Алгоритм обчислення відстані Дамерау-Левенштейна

Теоретичні відомості

Відстань Дамерау-Левенштейна є розширенням класичної відстані Левенштейна, яка додатково враховує операцію транспозиції сусідніх символів. Ця метрика була запропонована Фредеріком Дамерау в 1964 році на основі спостереження, що близько 80% всіх орфографічних помилок людини можна виправити однією з чотирьох операцій: вставкою, видаленням, заміною або транспозицією сусідніх символів.

Формально, відстань Дамерау-Левенштейна між двома рядками визначається як мінімальна кількість операцій, необхідних для перетворення одного рядка в інший, де дозволені такі операції:

- Вставка одного символу

- Видалення одного символу

- Заміна одного символу на інший

- Транспозиція (перестановка) двох сусідніх символів

Для двох рядків s_1 довжиною m та s_2 довжиною n, визначимо D[i,j] як відстань Дамерау-Левенштейна між префіксами $s_1[1..i]$ та $s_2[1..j]$.

Рекурентне співвідношення має вигляд:

```
D[0,0] = 0
D[i,0] = i для всіх i від 1 до m
D[0,j] = j для всіх j від 1 до n

D[i,j] = min {
    D[i-1,j] + 1,                   // видалення
    D[i,j-1] + 1,                   // вставка
    D[i-1,j-1] + cost(i,j),         // заміна або збіг
    D[i-2,j-2] + 1                  // транспозиція (якщо можлива)
}

де cost(i,j) = 0, якщо s₁[i] = s₂[j]
               1, якщо s₁[i] ≠ s₂[j]
```

Транспозиція можлива, якщо:
i > 1 та j > 1 та $s_1[i] = s_2[j-1]$ та $s_1[i-1] = s_2[j]$
Важливо розрізняти дві версії алгоритму:

1. **Відстань оптимального вирівнювання рядків** - простіша версія, де кожна підпослідовність може бути редагована не більше одного разу

2. **Справжня відстань Дамерау-Левенштейна** - дозволяє множинні редагування однієї підпослідовності

Більшість реалізацій використовують першу версію через її простоту та ефективність.

Алгоритм оптимального вирівнювання рядків

```
Алгоритм OptimalStringAlignment(s1, s2):
    Вхід:
        s1 - перший рядок довжиною m
        s2 - другий рядок довжиною n
    Вихід:
        distance - відстань Дамерау-Левенштейна

    m ← довжина(s1)
    n ← довжина(s2)

    // Створюємо матрицю для динамічного програмування
    D ← новий масив[0..m, 0..n]

    // Ініціалізація базових випадків
    для i від 0 до m:
        D[i,0] ← i

    для j від 0 до n:
        D[0,j] ← j

    // Заповнення матриці
    для i від 1 до m:
        для j від 1 до n:
            // Вартість заміни
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← 1

            // Базові три операції
            deletion ← D[i-1,j] + 1
```

```
            insertion ← D[i,j-1] + 1
            substitution ← D[i-1,j-1] + cost

            D[i,j] ← min(deletion, insertion, substitution)

            // Перевірка можливості транспозиції
            якщо i > 1 та j > 1:
                якщо s1[i-1] = s2[j-2] та s1[i-2] = s2[j-1]:
                    transposition ← D[i-2,j-2] + 1
                    D[i,j] ← min(D[i,j], transposition)

    повернути D[m,n]
```

Справжня відстань Дамерау-Левенштейна

Для обчислення справжньої відстані Дамерау-Левенштейна потрібен складніший алгоритм:

```
Алгоритм TrueDamerauLevenshtein(s1, s2):
    Вхід:
        s1, s2 - вхідні рядки
    Вихід:
        distance - справжня відстань Дамерау-Левенштейна

    m ← довжина(s1)
    n ← довжина(s2)

    // Визначаємо розмір алфавіту
    alphabet ← унікальні символи з s1 та s2
    alphabetSize ← розмір(alphabet)

    // Створюємо розширену матрицю
    INF ← m + n + 1
    H ← новий масив[-1..m, -1..n]

    // Ініціалізація
    H[-1,-1] ← INF
    для i від 0 до m:
        H[i,-1] ← INF
    для j від 0 до n:
        H[-1,j] ← INF

    // Базові випадки
    H[0,0] ← 0
    для i від 1 до m:
        H[i,0] ← i
    для j від 1 до n:
```

```
        H[0,j] ← j

    // Словник для зберігання останньої позиції кожного символу
    DA ← новий словник з початковим значенням 0 для всіх символів

    // Основний цикл
    для i від 1 до m:
        DB ← 0  // Остання позиція в s2

        для j від 1 до n:
            // Позиція останнього входження s2[j] в s1
            i1 ← DA[s2[j-1]]
            // Позиція останнього входження s1[i] в s2
            j1 ← DB

            // Вартість заміни
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
                DB ← j
            інакше:
                cost ← 1

            // Мінімум з усіх можливих операцій
            H[i,j] ← min(
                H[i-1,j] + 1,           // видалення
                H[i,j-1] + 1,           // вставка
                H[i-1,j-1] + cost,      // заміна
                H[i1-1,j1-1] + (i-i1-1) + 1 + (j-j1-1)  // транспозиц
ія
            )

        // Оновлюємо останню позицію s1[i] в DA
        DA[s1[i-1]] ← i

    повернути H[m,n]
```

Розглянемо детальний приклад обчислення відстані між словами “CA” та “ABC”:

Крок 1: Ініціалізація матриці

```
     ""  A  B  C
  ""   0  1  2  3
  C    1
  A    2
```

Крок 2: Заповнення позиції D[1,1] (порівнюємо “C” з “A”)

- s1[0] = ‘C’, s2[0] = ‘A’ → різні символи, cost = 1
- Видалення: D[0,1] + 1 = 1 + 1 = 2
- Вставка: D[1,0] + 1 = 1 + 1 = 2
- Заміна: D[0,0] + 1 = 0 + 1 = 1
- Транспозиція: неможлива (i=1, потрібно i>1)
- D[1,1] = min(2, 2, 1) = 1

Крок 3: Заповнення позиції D[1,2] (порівнюємо “C” з “AB”)
- s1[0] = ‘C’, s2[1] = ‘B’ → різні символи, cost = 1
- Видалення: D[0,2] + 1 = 2 + 1 = 3
- Вставка: D[1,1] + 1 = 1 + 1 = 2
- Заміна: D[0,1] + 1 = 1 + 1 = 2
- Транспозиція: неможлива
- D[1,2] = min(3, 2, 2) = 2

Крок 4: Заповнення позиції D[1,3] (порівнюємо “C” з “ABC”)
- s1[0] = ‘C’, s2[2] = ‘C’ → однакові символи, cost = 0
- Видалення: D[0,3] + 1 = 3 + 1 = 4
- Вставка: D[1,2] + 1 = 2 + 1 = 3
- Заміна: D[0,2] + 0 = 2 + 0 = 2
- Транспозиція: неможлива
- D[1,3] = min(4, 3, 2) = 2

Крок 5: Заповнення позиції D[2,1] (порівнюємо “CA” з “A”)
- s1[1] = ‘A’, s2[0] = ‘A’ → однакові символи, cost = 0
- Видалення: D[1,1] + 1 = 1 + 1 = 2
- Вставка: D[2,0] + 1 = 2 + 1 = 3
- Заміна: D[1,0] + 0 = 1 + 0 = 1
- Транспозиція: неможлива
- D[2,1] = min(2, 3, 1) = 1

Крок 6: Заповнення позиції D[2,2] (порівнюємо “CA” з “AB”)
- s1[1] = ‘A’, s2[1] = ‘B’ → різні символи, cost = 1
- Видалення: D[1,2] + 1 = 2 + 1 = 3

- Вставка: D[2,1] + 1 = 1 + 1 = 2

- Заміна: D[1,1] + 1 = 1 + 1 = 2

- Транспозиція: неможлива (символи не відповідають умові)

- D[2,2] = min(3, 2, 2) = 2

Крок 7: Заповнення позиції D[2,3] (порівнюємо “CA” з “ABC”)

- s1[1] = ‘A’, s2[2] = ‘C’ → різні символи, cost = 1

- Видалення: D[1,3] + 1 = 2 + 1 = 3

- Вставка: D[2,2] + 1 = 2 + 1 = 3

- Заміна: D[1,2] + 1 = 2 + 1 = 3

- Транспозиція: перевіряємо умову

    - s1[1] = ‘A’ повинно дорівнювати s2[1] = ‘B’ → ні

    - s1[0] = ‘C’ повинно дорівнювати s2[2] = ‘C’ → так

    - Умова не виконується повністю

- D[2,3] = min(3, 3, 3) = 3

Фінальна матриця:

	""	A	B	C
""	0	1	2	3
C	1	1	2	2
A	2	1	2	3

Відстань Дамерау-Левенштейна між “CA” та “ABC” дорівнює 3.

Відновлення послідовності операцій

```
Алгоритм GetDamerauOperations(D, s1, s2):
    Вхід:
        D - заповнена матриця відстаней
        s1, s2 - вхідні рядки
    Вихід:
        operations - список операцій редагування

    operations ← порожній список
    i ← довжина(s1)
    j ← довжина(s2)

    поки i > 0 або j > 0:
        якщо i = 0:
            додати "Вставити s2[j]" до operations
            j ← j - 1
```

```
        інакше якщо j = 0:
            додати "Видалити s1[i]" до operations
            i ← i - 1
        інакше:
            current ← D[i,j]

            // Перевіряємо транспозицію
            якщо i > 1 та j > 1:
                якщо s1[i-1] = s2[j-2] та s1[i-2] = s2[j-1]:
                    якщо current = D[i-2,j-2] + 1:
                        додати "Транспозиція s1[i-1] та s1[i]" до ope
rations
                        i ← i - 2
                        j ← j - 2
                        продовжити цикл

            // Перевіряємо інші операції
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← 1

            якщо current = D[i-1,j-1] + cost:
                якщо cost = 0:
                    додати "Залишити s1[i]" до operations
                інакше:
                    додати "Замінити s1[i] на s2[j]" до operations
                i ← i - 1
                j ← j - 1
            інакше якщо current = D[i-1,j] + 1:
                додати "Видалити s1[i]" до operations
                i ← i - 1
            інакше:
                додати "Вставити s2[j]" до operations
                j ← j - 1

    розвернути operations
    повернути operations
```

Оптимізація використання пам’яті

Для обчислення лише відстані можна використовувати три рядки замість повної матриці:

```
Алгоритм DamerauLevenshteinOptimized(s1, s2):
    Вхід:
        s1, s2 - вхідні рядки
```

```
    Вихід:
        distance - відстань Дамерау-Левенштейна

    m ← довжина(s1)
    n ← довжина(s2)

    // Три рядки для зберігання значень
    prevPrevRow ← новий масив[0..n]
    prevRow ← новий масив[0..n]
    currRow ← новий масив[0..n]

    // Ініціалізація першого рядка
    для j від 0 до n:
        prevRow[j] ← j

    для i від 1 до m:
        currRow[0] ← i

        для j від 1 до n:
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← 1

            deletion ← prevRow[j] + 1
            insertion ← currRow[j-1] + 1
            substitution ← prevRow[j-1] + cost

            currRow[j] ← min(deletion, insertion, substitution)

            // Перевірка транспозиції
            якщо i > 1 та j > 1:
                якщо s1[i-1] = s2[j-2] та s1[i-2] = s2[j-1]:
                    transposition ← prevPrevRow[j-2] + 1
                    currRow[j] ← min(currRow[j], transposition)

        // Зсув рядків
        temp ← prevPrevRow
        prevPrevRow ← prevRow
        prevRow ← currRow
        currRow ← temp

    повернути prevRow[n]
```

Варіації та модифікації

1. Зважена відстань Дамерау-Левенштейна

```
Алгоритм WeightedDamerauLevenshtein(s1, s2, wIns, wDel, wSub, wTrans)
:
    Вхід:
        s1, s2 - рядки
        wIns, wDel, wSub, wTrans - ваги операцій
    Вихід:
        distance - зважена відстань

    m ← довжина(s1)
    n ← довжина(s2)
    D ← новий масив[0..m, 0..n]

    // Ініціалізація з вагами
    D[0,0] ← 0
    для i від 1 до m:
        D[i,0] ← D[i-1,0] + wDel
    для j від 1 до n:
        D[0,j] ← D[0,j-1] + wIns

    для i від 1 до m:
        для j від 1 до n:
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← wSub

            D[i,j] ← min(
                D[i-1,j] + wDel,
                D[i,j-1] + wIns,
                D[i-1,j-1] + cost
            )

            якщо i > 1 та j > 1:
                якщо s1[i-1] = s2[j-2] та s1[i-2] = s2[j-1]:
                    D[i,j] ← min(D[i,j], D[i-2,j-2] + wTrans)

    повернути D[m,n]
```

Аналіз складності

Часова складність:

- Алгоритм оптимального вирівнювання: O(m × n)

- Справжня відстань Дамерау-Левенштейна: O(m × n × max(m, n))

Просторова складність:

- Повна матриця: O(m × n)

- Оптимізована версія: O(min(m, n))

- Справжня відстань: O(m × n + |Σ|), де |Σ| - розмір алфавіту

Особливості реалізації

Важливі моменти при реалізації алгоритму:

1. Індексація: У псевдокоді використовується індексація з 1 для рядків, але з 0 для масивів. При реалізації потрібно бути уважним з індексами.

2. Перевірка меж: Для транспозиції потрібна перевірка i > 1 та j > 1, щоб уникнути виходу за межі масиву.

3. Порядок операцій: При відновленні шляху важливо перевіряти транспозицію першою, оскільки вона змінює два індекси одночасно.

Область застосування

Алгоритм Дамерау-Левенштейна широко використовується в:

- Виправлення орфографічних помилок - особливо ефективний для помилок типу “teh” → “the”

- Системи автодоповнення - пропозиція правильних варіантів при введенні тексту

- Біоінформатика - аналіз генетичних мутацій з врахуванням інверсій

- Обробка природної мови - нормалізація тексту та виявлення варіацій слів

- Дедуплікація даних - знаходження майже ідентичних записів у базах даних

- Розпізнавання тексту (OCR) - корекція помилок розпізнавання

- Пошукові системи - обробка запитів з помилками

- Системи контролю якості - виявлення помилок введення даних

Приклади застосування

Приклад 1: Виправлення типових помилок друку

```
Введено: "recieve"
Словник: ["receive", "recipe", "receipt", "deceive"]

Відстані Дамерау-Левенштейна:
- "recieve" → "receive": 1 (транспозиція ei → ie)
- "recieve" → "recipe": 3
```

```
- "recieve" → "receipt": 3
- "recieve" → "deceive": 2

Найкраще виправлення: "receive"
```

Приклад 2: Аналіз помилок при введенні номерів

```
Правильний номер: "12345"
Введений номер: "13245"

Відстань Дамерау-Левенштейна: 1
Тип помилки: транспозиція цифр 2 та 3
```

Приклад 3: Пошук схожих імен у базі даних

```
Пошуковий запит: "Johanson"
База даних: ["Johnson", "Johansson", "Johansen", "Johnston"]

Результати з відстанню ≤ 2:
- "Johnson": 1 (транспозиція)
- "Johansson": 1 (вставка)
- "Johansen": 2 (заміна + видалення)
```

Переваги та недоліки

Переваги

- Врахування транспозицій - природна обробка найпоширенішої помилки при введенні тексту

- Висока точність - краще моделює реальні помилки людини

- Ефективність для коротких рядків - оптимальна для виправлення слів

- Метричні властивості - версія оптимального вирівнювання є метрикою

- Інтуїтивність - результати відповідають людським очікуванням

- Широка застосовність - підходить для різних типів даних

Недоліки

- Складність справжньої версії - алгоритм для справжньої відстані складніший

- Більше використання пам’яті - потрібно зберігати додаткові рядки для транспозицій

- Не завжди метрика - справжня відстань не задовольняє нерівність трикутника

- Обмеження на транспозиції - враховує лише сусідні символи

- Складність паралелізації - залежності між клітинками ускладнюють розпаралелювання

- Не оптимальна для довгих текстів - квадратична складність

Контрольні запитання

1. Чим відрізняється відстань Дамерау-Левенштейна від класичної відстані Левенштейна?

2. Які умови необхідні для виконання операції транспозиції?

3. В чому різниця між відстанню оптимального вирівнювання та справжньою відстанню Дамерау-Левенштейна?

4. Яка часова складність алгоритму оптимального вирівнювання?

5. Чому транспозиція важлива для виправлення орфографічних помилок?

6. Як модифікувати алгоритм для різних ваг операцій?

7. Скільки рядків матриці потрібно зберігати для оптимізованої версії?

8. Чи є відстань оптимального вирівнювання метрикою?

9. Для яких застосувань критично важливо враховувати транспозиції?

10. Як відновити послідовність операцій з заповненої матриці?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму обчислення відстані Дамерау-Левенштейна

2. Мета

Вивчити розширення класичної редакційної відстані для врахування транспозицій, реалізувати різні версії алгоритму Дамерау-Левенштейна, дослідити їх ефективність для задач виправлення помилок.

3. Завдання

- Реалізувати алгоритм оптимального вирівнювання рядків

- Реалізувати справжню відстань Дамерау-Левенштейна

- Створити функцію відновлення послідовності операцій

- Розробити оптимізовану версію з обмеженим використанням пам’яті

- Реалізувати зважену версію алгоритму

- Створити візуалізацію процесу обчислення

- Розробити систему виправлення помилок на основі алгоритму

- Провести порівняльний аналіз з класичною відстанню Левенштейна

4. Опис

Лабораторна робота спрямована на глибоке вивчення алгоритму Дамерау-Левенштейна через практичну реалізацію його різних варіантів. Студенти мають дослідити вплив врахування транспозицій на якість виправлення помилок, порівняти різні версії алгоритму та створити практичні застосування.

5. Порядок виконання

- Створити клас DamerauLevenshteinCalculator

- Реалізувати метод calculateOSA для відстані оптимального вирівнювання

- Імплементувати метод calculateTrue для справжньої відстані

- Додати метод getOperations для відновлення операцій

- Створити метод calculateOptimized з оптимізацією пам’яті

- Реалізувати візуалізатор DamerauMatrixVisualizer

- Розробити клас TypoCorrector для виправлення помилок

- Створити набір тестів для різних випадків

- Провести експерименти з реальними даними

6. Результат

Програма повинна: - Коректно обчислювати обидві версії відстані Дамерау-Левенштейна - Виявляти та візуалізувати транспозиції - Відновлювати оптимальну послідовність операцій - Ефективно працювати для практичних задач - Демонструвати переваги врахування транспозицій - Надавати детальну статистику операцій

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас DamerauLevenshteinCalculator з усіма методами

- Метод calculateOSA для оптимального вирівнювання

- Метод calculateTrue для справжньої відстані

- Метод getOperations для відновлення шляху

- Клас TypoCorrector для практичного застосування

- Модуль візуалізації матриці та операцій
- Модуль порівняльного аналізу алгоритмів

2. Продемонструвати роботу додатку:
- Обчислення відстані для різних пар рядків
- Виявлення та демонстрація транспозицій
- Порівняння результатів двох версій алгоритму
- Візуалізація процесу заповнення матриці
- Робота системи виправлення помилок
- Статистика типів операцій для реальних даних

3. Бути готовим відповісти на питання:
- Як визначається можливість транспозиції?
- Чому потрібна перевірка i > 1 та j > 1?
- В яких випадках результати двох версій відрізняються?
- Як вибрати оптимальні ваги для конкретної задачі?
- Які типи помилок найкраще виправляються алгоритмом?

Тести

1. Відстань Дамерау-Левенштейна між “CA” та “AC” дорівнює:

1) 0

2) 1

3) 2

4) 3

2. Яку додаткову операцію враховує алгоритм Дамерау-Левенштейна порівняно з класичним Левенштейном?

1) Копіювання

2) Переміщення

3) Транспозицію

4) Дублювання

3. Для виконання транспозиції на позиції (i,j) необхідно, щоб:

1) i ⩾ 1 та j ⩾ 1

2) i > 1 та j > 1

3) $i > 0$ та $j > 0$

4) $i \geqslant 2$ та $j \geqslant 2$

4. Часова складність алгоритму оптимального вирівнювання:

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) O(m × n)

5. Скільки рядків матриці необхідно для оптимізованої версії алгоритму?

1) 1

2) 2

3) 3

4) 4

Відповіді:

1. b)

2. c)

3. b)

4. d)

5. c)

Тести підвищеної складності

1. Відстань Дамерау-Левенштейна між “abcd” та “acbd” за алгоритмом оптимального вирівнювання:

1) 1

2) 2

3) 3

4) 4

2. Для рядків “ABC” та “CBA” різниця між справжньою відстанню та оптимальним вирівнюванням:

1) 0

2) 1

3) 2

4) 3

3. При якій умові відстань оптимального вирівнювання НЕ є метрикою?

1) Завжди є метрикою

2) Для рядків з повтореннями

3) Ніколи не є метрикою

4) Для Unicode рядків

4. Максимальна кількість транспозицій для перетворення рядка довжини n:

1) n/2

2) n-1

3) n

4) n^2

5. Просторова складність справжньої відстані Дамерау-Левенштейна:

1) O(n)

2) $O(n^2)$

3) $O(n^2 + |\Sigma|)$

4) $O(n^3)$

6. Для яких рядків транспозиція дає найбільшу перевагу?

1) Анаграми

2) Паліндроми

3) Рядки з переставленими сусідніми символами

4) Випадкові рядки

7. При обчисленні D[3,3] скільки попередніх значень потрібно перевірити?

1) 3

2) 4

3) 5

4) 6

8. Яка операція НЕ може бути частиною оптимального шляху одночасно з транспозицією тих самих позицій?

1) Вставка

2) Видалення

3) Заміна цих символів

4) Всі можуть

9. Для паралелізації алгоритму найкраще обчислювати:

1) По рядках

2) По стовпцях

3) По діагоналях

4) По блоках 2×2

10. При якій вазі транспозиції алгоритм стає еквівалентним Левенштейну?

1) 0

2) 1

3) 2

4) ∞

Відповіді:

1. b) транспозиція bc→cb вважається як дві операції в оптимальному вирівнюванні

2. b) справжня відстань враховує складніші перестановки

3. a) Завжди є метрикою - версія оптимального вирівнювання задовольняє всі аксіоми

4. a) максимум кожна пара сусідніх символів

5. c) матриця плюс словник для алфавіту

6. c) Транспозиція найефективніша для сусідніх перестановок

7. b) три базові операції плюс транспозиція

8. c) Заміна тих самих символів конфліктує з транспозицією

9. c) Елементи на діагоналі незалежні

10. d) нескінченна вага робить транспозицію неможливою

Завдання для самостійної роботи

1. Реалізуйте алгоритм Дамерау-Левенштейна для роботи з послідовностями токенів замість символів (наприклад, для порівняння речень).

2. Створіть адаптивний алгоритм, який автоматично визначає оптимальні ваги операцій на основі статистики помилок.

3. Розробіть розширення алгоритму для врахування транспозицій несусідніх символів з обмеженою відстанню.

4. Імплементуйте алгоритм для обчислення k найближчих сусідів у метричному просторі Дамерау-Левенштейна.

5. Створіть систему автоматичного виявлення типу помилки (друкарська, OCR, фонетична) на основі патернів редагувань.

6. Реалізуйте інкрементальну версію алгоритму для обчислення відстані при додаванні символів до рядка.

7. Розробіть алгоритм кластеризації рядків на основі відстані Дамерау-Левенштейна.

8. Створіть нейромережеву апроксимацію відстані для прискорення обчислень на великих даних.

9. Імплементуйте алгоритм для знаходження медіани множини рядків у метриці Дамерау-Левенштейна.

10. Розробіть систему нечіткого пошуку в базі даних з індексацією на основі відстані Дамерау-Левенштейна.

Теми лабораторних робіт

1. Розробка інтелектуальної клавіатури для мобільних пристроїв з виправленням помилок на основі Дамерау-Левенштейна.

2. Створення системи валідації та очищення даних для CRM з виявленням дублікатів та помилок введення.

3. Імплементація модуля перевірки правопису для текстового редактора з врахуванням контексту.

4. Розробка системи розпізнавання та виправлення помилок у відсканованих документах.

5. Створення інструменту для аналізу та виправлення помилок у лог-файлах серверів.

6. Реалізація системи пошуку схожих товарів в e-commerce за неточними назвами.

7. Розробка модуля автодоповнення для IDE з корекцією помилок у назвах змінних та функцій.

8. Створення системи виявлення фішингових доменів на основі схожості з легітимними.

9. Імплементація інструменту для злиття та дедуплікації контактів у адресній книзі.

10. Розробка системи контролю якості введення медичних даних з виявленням критичних помилок.

10.2.3. Алгоритм Гіршберга для обчислення редакційної відстані

Теоретичні відомості

Алгоритм Гіршберга, розроблений Деном Гіршбергом у 1975 році, є елегантним рішенням проблеми високої просторової складності класичних алгоритмів обчислення редакційної відстані. Класичний алгоритм Левенштейна потребує O(m×n) пам'яті для зберігання матриці динамічного програмування, що стає критичним обмеженням при роботі з довгими послідовностями. Алгоритм Гіршберга вирішує цю проблему, зменшуючи просторову складність до O(min(m,n)), зберігаючи при цьому можливість відновлення оптимальної послідовності операцій редагування.

Ключова ідея алгоритму базується на техніці "розділяй і володарюй" та властивості оптимальності підзадач у динамічному програмуванні. Алгоритм розділяє задачу на дві підзадачі меншого розміру, рекурсивно їх розв'язує та об'єднує результати.

Нехай маємо два рядки s_1 довжиною m та s_2 довжиною n. Для будь-якого індексу k ($0 \leqslant k \leqslant n$), оптимальне вирівнювання рядків s_1 та s_2 можна розділити на дві частини: - Оптимальне вирівнювання $s_1[1..i]$ з $s_2[1..k]$ - Оптимальне вирівнювання $s_1[i+1..m]$ з $s_2[k+1..n]$

для деякого i, де $0 \leqslant i \leqslant m$.

Ця властивість дозволяє рекурсивно розбивати задачу на менші підзадачі.

Прямий та зворотний напрямки обчислення

Для знаходження точки розділу, алгоритм обчислює редакційні відстані в двох напрямках:

- Прямий напрямок: від початку рядків до середини

- Зворотний напрямок: від кінця рядків до середини

Базові компоненти алгоритму

Обчислення відстані з обмеженою пам'яттю

```
Алгоритм ComputeDistanceLinearSpace(s1, s2):
    Вхід:
```

```
        s1 - перший рядок довжиною m
        s2 - другий рядок довжиною n
    Вихід:
        lastRow - останній рядок матриці відстаней

    m ← довжина(s1)
    n ← довжина(s2)

    // Використовуємо два рядки замість повної матриці
    prevRow ← новий масив[0..n]
    currRow ← новий масив[0..n]

    // Ініціалізація першого рядка
    для j від 0 до n:
        prevRow[j] ← j

    // Обчислення по рядках
    для i від 1 до m:
        currRow[0] ← i

        для j від 1 до n:
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← 1

            deletion ← prevRow[j] + 1
            insertion ← currRow[j-1] + 1
            substitution ← prevRow[j-1] + cost

            currRow[j] ← min(deletion, insertion, substitution)

        // Обмін рядків
        поміняти місцями prevRow та currRow

    повернути prevRow
```

Обчислення середнього рядка матриці

```
Алгоритм ComputeMiddleRow(s1, s2):
    Вхід:
        s1 - рядок довжиною m
        s2 - рядок довжиною n
    Вихід:
        middleRow - рядок відстаней для середини s2
```

```
    m ← довжина(s1)
    n ← довжина(s2)
    mid ← n / 2  // ціла частина

    // Обчислюємо відстані від початку до середини
    prefix ← s2[0..mid-1]
    forwardRow ← ComputeDistanceLinearSpace(s1, prefix)

    // Обчислюємо відстані від кінця до середини
    suffix ← s2[mid..n-1]
    s1Rev ← реверс(s1)
    suffixRev ← реверс(suffix)
    backwardRow ← ComputeDistanceLinearSpace(s1Rev, suffixRev)

    // Реверсуємо результат зворотного обчислення
    backwardRow ← реверс(backwardRow)

    // Об'єднуємо результати
    middleRow ← новий масив[0..m]
    для i від 0 до m:
        middleRow[i] ← forwardRow[i] + backwardRow[i]

    повернути middleRow
```

Основний алгоритм Гіршберга

```
Алгоритм Hirschberg(s1, s2):
    Вхід:
        s1 - перший рядок довжиною m
        s2 - другий рядок довжиною n
    Вихід:
        alignment - оптимальне вирівнювання рядків

    m ← довжина(s1)
    n ← довжина(s2)

    // Базові випадки
    якщо m = 0:
        повернути вирівнювання з n вставками символів s2
    якщо n = 0:
        повернути вирівнювання з m видаленнями символів s1
    якщо m = 1:
        повернути OptimalAlignmentOneRow(s1, s2)
    якщо n = 1:
        повернути OptimalAlignmentOneColumn(s1, s2)
```

```
    // Знаходимо точку розділу
    mid ← n / 2
    splitPoint ← FindSplitPoint(s1, s2, mid)

    // Рекурсивно розв'язуємо підзадачі
    leftAlignment ← Hirschberg(s1[0..splitPoint-1], s2[0..mid-1])
    rightAlignment ← Hirschberg(s1[splitPoint..m-1], s2[mid..n-1])

    // Об'єднуємо результати
    повернути конкатенація(leftAlignment, rightAlignment)
```

Знаходження точки розділу

```
Алгоритм FindSplitPoint(s1, s2, mid):
    Вхід:
        s1 - перший рядок довжиною m
        s2 - другий рядок довжиною n
        mid - точка розділу s2
    Вихід:
        splitPoint - оптимальна точка розділу s1

    m ← довжина(s1)
    n ← довжина(s2)

    // Обчислюємо відстані в прямому напрямку
    prefix ← s2[0..mid-1]
    forwardRow ← ComputeForwardDistances(s1, prefix)

    // Обчислюємо відстані в зворотному напрямку
    suffix ← s2[mid..n-1]
    backwardRow ← ComputeBackwardDistances(s1, suffix)

    // Знаходимо оптимальну точку розділу
    minCost ← нескінченність
    splitPoint ← 0

    для i від 0 до m:
        totalCost ← forwardRow[i] + backwardRow[m-i]
        якщо totalCost < minCost:
            minCost ← totalCost
            splitPoint ← i

    повернути splitPoint
```

Обчислення відстаней у прямому напрямку

```
Алгоритм ComputeForwardDistances(s1, s2):
    Вхід:
```

```
        s1 - перший рядок
        s2 - префікс другого рядка
    Вихід:
        distances - масив відстаней

    m ← довжина(s1)
    n ← довжина(s2)

    prevRow ← новий масив[0..n]
    currRow ← новий масив[0..n]

    // Ініціалізація
    для j від 0 до n:
        prevRow[j] ← j

    // Зберігаємо всі проміжні результати
    distances ← новий масив[0..m]
    distances[0] ← prevRow[n]

    для i від 1 до m:
        currRow[0] ← i

        для j від 1 до n:
            якщо s1[i-1] = s2[j-1]:
                cost ← 0
            інакше:
                cost ← 1

            currRow[j] ← min(
                prevRow[j] + 1,      // видалення
                currRow[j-1] + 1,    // вставка
                prevRow[j-1] + cost  // заміна
            )

        distances[i] ← currRow[n]
        поміняти місцями prevRow та currRow

    повернути distances
```

Обчислення відстаней у зворотному напрямку

```
Алгоритм ComputeBackwardDistances(s1, s2):
    Вхід:
        s1 - перший рядок
        s2 - суфікс другого рядка
    Вихід:
```

```
        distances - масив відстаней від кінця

    // Реверсуємо рядки
    s1Rev ← реверс(s1)
    s2Rev ← реверс(s2)

    // Обчислюємо відстані для реверсованих рядків
    forwardDistances ← ComputeForwardDistances(s1Rev, s2Rev)

    // Реверсуємо результат
    повернути реверс(forwardDistances)
```

Обробка базових випадків

Вирівнювання з одним рядком

```
Алгоритм OptimalAlignmentOneRow(s1, s2):
    Вхід:
        s1 - рядок довжиною 1
        s2 - рядок довжиною n
    Вихід:
        alignment - оптимальне вирівнювання

    char ← s1[0]
    alignment ← порожній список

    // Шукаємо перше входження char в s2
    found ← false
    для j від 0 до n-1:
        якщо s2[j] = char і не found:
            додати "MATCH" до alignment
            found ← true
        інакше:
            додати "INSERT s2[j]" до alignment

    якщо не found:
        // Замінюємо перший символ або видаляємо та вставляємо всі
        якщо n > 0:
            alignment[0] ← "SUBSTITUTE s1[0] with s2[0]"
        інакше:
            додати "DELETE s1[0]" до alignment

    повернути alignment
```

Розглянемо детальний приклад роботи алгоритму для рядків s1 = “AGCT” та s2 = “ACGT”.

Крок 1: Початковий виклик

- m = 4, n = 4

- Жоден з базових випадків не виконується - mid = n / 2 = 2

Крок 2: Обчислення відстаней у прямому напрямку Для префікса “AC” рядка s2:

```
    ""  A  C
""   0  1  2
A    1  0  1
G    2  1  1
C    3  2  1
T    4  3  2
```

forwardRow = [2, 1, 1, 2]

Крок 3: Обчислення відстаней у зворотному напрямку Для суфікса “GT” рядка s2:

```
    ""  T  G
""   0  1  2
T    1  0  1
C    2  1  1
G    3  2  1
A    4  3  2
```

backwardRow = [2, 1, 1, 2] (після реверсу)

Крок 4: Знаходження точки розділу

Обчислюємо суми:

- i=0: forward[0] + backward[4] = 2 + 2 = 4 - i=1: forward[1] + backward[3] = 1 + 1 = 2

- i=2: forward[2] + backward[2] = 1 + 1 = 2

- i=3: forward[3] + backward[1] = 2 + 1 = 3

- i=4: forward[4] + backward[0] = 2 + 2 = 4

Мінімум досягається при i=1 або i=2. Обираємо splitPoint = 2.

Крок 5: Рекурсивні виклики

- Лівий виклик: Hirschberg(“AG”, “AC”)

- Правий виклик: Hirschberg(“CT”, “GT”)

Крок 6: Об’єднання результатів Фінальне вирівнювання: A→A (match), G→C (substitute), C→G (substitute), T→T (match)

Аналіз складності

Часова складність

Нехай T(m,n) - час роботи алгоритму для рядків довжиною m та n.

Рекурентне співвідношення:

$T(m,n) = O(mn) + T(m_1, n/2) + T(m_2, n/2)$
де $m_1 + m_2 = m$.

Розв'язуючи рекурентне співвідношення, отримуємо: T(m,n) = O(mn × log min(m,n))

Це гірше за O(mn) класичного алгоритму, але різниця логарифмічна.

Просторова складність

Ключова перевага алгоритму: - На кожному рівні рекурсії використовується O(min(m,n)) пам'яті - Глибина рекурсії O(log min(m,n)) - Загальна просторова складність: O(min(m,n) × log min(m,n))

Для великих послідовностей це суттєва економія порівняно з O(mn).

Оптимізації алгоритму

1. Вибір напрямку розділу

```
Алгоритм ChooseSplitDirection(m, n):
    якщо m < n:
        повернути "горизонтальний"
    інакше:
        повернути "вертикальний"
```

Розділяємо довший рядок для мінімізації глибини рекурсії.

2. Кешування проміжних результатів

```
Алгоритм HirschbergWithCache(s1, s2, cache):
    key ← (s1, s2)
    якщо key в cache:
        повернути cache[key]

    result ← Hirschberg(s1, s2)
    cache[key] ← result
    повернути result
```

3. Паралелізація

Рекурсивні виклики незалежні, тому можна виконувати їх паралельно:

```
Алгоритм ParallelHirschberg(s1, s2):
    // ... знаходження точки розділу ...

    // Паралельне виконання
    паралельно:
        leftAlignment ← Hirschberg(s1[0..splitPoint-1], s2[0..mid-1])
```

```
        rightAlignment ← Hirschberg(s1[splitPoint..m-1], s2[mid..n-1]
)

    повернути конкатенація(leftAlignment, rightAlignment)
```

Область застосування

Алгоритм Гіршберга особливо корисний у таких сферах:

- Біоінформатика - вирівнювання довгих генетичних послідовностей (ДНК, РНК, білків)

- Обробка великих текстів - порівняння книг, документів, кодових баз

- Системи контролю версій - обчислення різниці між великими файлами

- Вбудовані системи - робота з обмеженою пам’яттю

- Мобільні пристрої - економія пам’яті та батареї

- Веб-сервіси - обробка великих даних без перевантаження сервера

- Аналіз логів - порівняння великих файлів журналів

- Дедуплікація даних - знаходження схожих великих файлів

Приклади застосування

Приклад 1: Порівняння геномів

При порівнянні двох бактеріальних геномів розміром по 5 мільйонів нуклеотидів: - Класичний алгоритм: 25 трильйонів клітинок пам’яті - Алгоритм Гіршберга: ~5 мільйонів клітинок пам’яті - Економія: в 5 мільйонів разів

Приклад 2: Аналіз змін у коді

Порівняння двох версій великого програмного проекту: - Файл 1: 100,000 рядків коду - Файл 2: 102,000 рядків коду - Алгоритм знаходить точні зміни з мінімальним використанням пам’яті

Приклад 3: Синхронізація документів

Синхронізація великих текстових документів між пристроями: - Документ на сервері: 50 МБ - Локальна копія: 49.8 МБ - Алгоритм ефективно знаходить різницю для передачі лише змін

Переваги та недоліки

Переваги

- Економія пам’яті - O(min(m,n)) замість O(mn) для відновлення шляху

- Масштабованість - можливість роботи з дуже довгими послідовностями

- Точність - знаходить оптимальне вирівнювання, як і класичний алгоритм
- Паралелізація - природна можливість розпаралелювання
- Універсальність - працює з будь-якими функціями вартості
- Елегантність - красиве застосування принципу “розділяй і володарюй”

Недоліки

- Складність реалізації - складніший за класичний алгоритм
- Додатковий час - O(log min(m,n)) додатковий множник у часі
- Накладні витрати рекурсії - стек викликів та повторні обчислення
- Складність налагодження - рекурсивна природа ускладнює відладку
- Неефективність для коротких рядків - накладні витрати перевищують вигоду
- Важко модифікувати - складніше адаптувати для спеціальних випадків

Контрольні запитання

1. В чому основна ідея алгоритму Гіршберга?
2. Яка просторова складність алгоритму порівняно з класичним підходом?
3. Чому алгоритм обчислює відстані в обох напрямках?
4. Як знаходиться оптимальна точка розділу?
5. Яка часова складність алгоритму Гіршберга?
6. Для яких задач алгоритм найбільш ефективний?
7. Чи можна застосувати алгоритм для відстані Дамерау-Левенштейна?
8. Як алгоритм обробляє базові випадки?
9. Які оптимізації можливі для алгоритму?
10. Чому важливо розділяти довший рядок?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Гіршберга для обчислення редакційної відстані з оптимальним використанням пам’яті

2. Мета

Вивчити принципи оптимізації просторової складності алгоритмів динамічного програмування, реалізувати алгоритм Гіршберга, дослідити його ефективність для великих послідовностей.

3. Завдання

- Реалізувати функцію обчислення відстані з лінійною пам’яттю
- Реалізувати функцію знаходження точки розділу
- Створити рекурсивний алгоритм Гіршберга
- Реалізувати відновлення оптимального вирівнювання
- Додати візуалізацію процесу розділення
- Порівняти використання пам’яті з класичним алгоритмом
- Дослідити вплив різних стратегій розділення
- Реалізувати паралельну версію алгоритму

4. Опис

Лабораторна робота спрямована на глибоке розуміння техніки “розділяй і володарюй” для оптимізації алгоритмів динамічного програмування. Студенти мають реалізувати повний алгоритм Гіршберга, дослідити його поведінку на різних даних та порівняти з класичними підходами.

5. Порядок виконання

- Створити клас HirschbergAlgorithm з основними методами
- Реалізувати метод computeDistanceLinearSpace
- Імплементувати метод findSplitPoint
- Створити рекурсивний метод computeAlignment
- Додати методи для обробки базових випадків
- Реалізувати візуалізатор HirschbergVisualizer
- Створити тести для різних розмірів даних
- Провести аналіз використання пам’яті
- Реалізувати паралельну версію

6. Результат

Програма повинна:

- Коректно обчислювати редакційну відстань для великих рядків
- Відновлювати оптимальне вирівнювання з мінімальною пам’яттю
- Візуалізувати процес рекурсивного розділення
- Демонструвати економію пам’яті порівняно з класичним алгоритмом

- Показувати статистику рекурсивних викликів
- Ефективно працювати з файлами великого розміру

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
- Клас HirschbergAlgorithm з усіма методами
- Метод computeDistanceLinearSpace для обчислення з обмеженою пам’яттю
- Метод findSplitPoint для знаходження оптимальної точки розділу
- Рекурсивний метод computeAlignment
- Методи обробки базових випадків
- Модуль візуалізації дерева рекурсивних викликів
- Модуль порівняння з класичним алгоритмом
2. Продемонструвати роботу додатку:
- Обчислення відстані для рядків різної довжини
- Візуалізація процесу розділення на підзадачі
- Відновлення оптимального вирівнювання
- Графік використання пам’яті залежно від розміру вхідних даних
- Порівняння часу виконання з класичним алгоритмом
- Робота з реальними файлами великого розміру
3. Бути готовим відповісти на питання:
- Як обчислюється точка розділу?
- Чому потрібні обчислення в обох напрямках?
- Як досягається економія пам’яті?
- Яка глибина рекурсії для конкретних прикладів?
- Як модифікувати алгоритм для інших метрик?

Тести

1. Просторова складність алгоритму Гіршберга для рядків довжини m та n:

1) O(mn)

2) O(m + n)

3) O(min(m,n))

4) O(min(m,n) × log min(m,n))

2. Алгоритм Гіршберга базується на техніці:

1) Жадібних алгоритмів

2) Розділяй і володарюй

3) Динамічного програмування знизу вгору

4) Пошуку з поверненням

3. Для знаходження точки розділу алгоритм обчислює відстані:

1) Тільки в прямому напрямку

2) Тільки в зворотному напрямку

3) В обох напрямках

4) Випадково в одному з напрямків

4. Часова складність алгоритму Гіршберга:

1) O(mn)

2) O(mn log n)

3) O(mn log min(m,n))

4) O(m + n)

5. Глибина рекурсії алгоритму Гіршберга:

1) O(m)

2) O(n)

3) O(mn)

4) O(log min(m,n))

Відповіді:

1. d)

2. b)

3. c)

4. c)

5. d)

Тести підвищеної складності

1. При розділі рядків довжини 100 та 200, скільки елементів обчислюється на першому рівні?

1) 10,000

2) 20,000

3) 40,000

4) 30,000

2. Яка операція НЕ може бути ефективно реалізована в алгоритмі Гіршберга?

1) Зважена редакційна відстань

2) Обмежена редакційна відстань

3) Відстань Дамерау-Левенштейна

4) Всі можуть бути реалізовані

3. При паралельній реалізації максимальний приріст швидкості:

1) O(log n)

2) O(n)

3) O(√n)

4) O(n/log n)

4. Для рядків однакової довжини n, скільки разів викликається базовий випадок?

1) n

2) 2n

3) n log n

4) n^2/log n

5. Яка структура даних найкраще підходить для кешування проміжних результатів?

1) Масив

2) Хеш-таблиця

3) Дерево

4) Черга

6. При якому співвідношенні довжин рядків алгоритм найменш ефективний?

1) m = n

2) m = 2n

3) $m = n^2$

4) $m = \log n$

7. Скільки пам’яті економиться для рядків довжини 10^6?

1) ~10^6 байт

2) ~10^9 байт

3) ~10^{12} байт

4) ~10^3 байт

8. Яка модифікація НЕ покращить продуктивність алгоритму?

1) Використання SIMD інструкцій

2) Збільшення розміру базового випадку

3) Випадковий вибір точки розділу

4) Кешування результатів

9. Для геномних послідовностей довжини 10^9, глибина рекурсії приблизно:

1) 10

2) 30

3) 100

4) 1000

10. При використанні алгоритму для файлів, оптимально розділяти:

1) По рядках

2) По блоках фіксованого розміру

3) По абзацах

4) Випадково

Відповіді:

1. b) обчислюється повна матриця 100×200

2. d) Всі варіанти можуть бути адаптовані для алгоритму Гіршберга

3. a) Обмежено глибиною дерева рекурсивних викликів

4. a) по одному для кожного листа дерева рекурсії

5. b) Хеш-таблиця забезпечує O(1) доступ для довільних підрядків

6. c) максимальна глибина рекурсії та дисбаланс навантаження

7. c) економія квадратичної пам’яті $O(n^2)$

8. c) Випадковий вибір погіршить якість розділення

9. b) $30 - \log_2(10^9) \approx 30$

10. a) По рядках для збереження семантичної цілісності

Завдання для самостійної роботи

1. Реалізуйте алгоритм Гіршберга для обчислення найдовшої спільної підпослідовності з відновленням самої підпослідовності.

2. Створіть адаптивну версію алгоритму, яка автоматично вибирає між класичним та Гіршберга залежно від доступної пам’яті.

3. Розробіть модифікацію для роботи з потоковими даними, коли рядки надходять частинами.

4. Імплементуйте версію для обчислення відстані між деревами з оптимальною пам’яттю.

5. Створіть гібридний алгоритм, який використовує GPU для малих підзадач.

6. Реалізуйте алгоритм для k-вимірних послідовностей (порівняння k рядків одночасно).

7. Розробіть версію з прогресивним уточненням результату для інтерактивних застосувань.

8. Створіть розподілену версію для кластерних обчислень великих геномів.

9. Імплементуйте алгоритм з підтримкою контрольних точок для відновлення після збоїв.

10. Розробіть версію для порівняння стиснутих даних без повної декомпресії.

10.4. Алгоритми стиснення

Уявімо, що перед нами завдання передати той самий зміст меншою кількістю бітів — без жодної втрати інформації. Це і є мета алгоритмів **безвтратного** стиснення: знайти та «упакувати» регулярності, які вже існують у даних, від повторів підряд до тонких статистичних закономірностей. У центрі стоїть проста ідея: якщо символи або цілі підрядки з’являються передбачувано, їх можна кодувати коротше, а рідкісні — довше.

Тому будь-який практичний компресор складається з двох взаємопов’язаних частин

– **моделювання** (як ми «передбачаємо» наступні символи та перетворюємо структуру даних у більш регулярний вигляд)

– **кодування** (як ми перетворюємо передбачувану послідовність у біти: Хаффман, арифметичне кодування тощо).

У цьому підрозділі ми познайомимося з чотирма фундаментальними сімействами методів, які сформували сучасну практику стиснення.

- **Перетворення Берроуза–Вілера (BWT)** Це не компресор «сам по собі», а оборотне перетворення, що переставляє символи так, щоб однакові символи і подібні контексти опинились поруч. Після BWT дані стають надзвичайно зручними для простих технік на кшталт RLE (run-length encoding), Move-To-Front і, зрештою, ентропійного кодування. Важливо, що BWT зберігає всю інформацію й легко відновлюється, але ключовий виграш приносить саме покращення локальності повторів.

– **LZ77** працює зі зсувним вікном: кожен новий фрагмент замінюється трійкою на кшталт (зсув, довжина, наступний символ), що вказує на минулий збіг у вже прочитаному тексті. Це зручно для потокової обробки.
– **LZ78** будує явний словник фраз, кожна нова фраза кодується індексом попередньої та новим символом. Різниця концепцій веде до різних компромісів щодо пам’яті, швидкості та співвідношення «стиснення/ресурси».

- **LZW** Варіант сімейства LZ, який автоматично і розумно розширює словник без явної передачі «наступного символу»: кодується лише індекс фрази. Історично LZW приніс популярність словниковим методам завдяки простоті декодування та непоганому коефіцієнту стиснення для текстів і індексованої графіки.

- Кодування Хаффмана і Фано – дуже схожі методи кодування, демонструють найкращі результати.

- **DEFLATE** Інженерне поєднання найкращого з двох світів: LZ77 для виявлення повторів і Хаффманівського кодування для ентропійного пакування отриманих пар (довжина, відстань) і літералів. Саме така композиція дає баланс між ефективністю, швидкістю та простотою впровадження, завдяки чому формат використовується в архівах і мережевих протоколах, а також у графічних контейнерах.

Подальші секції підрозділу послідовно розкриють кожен із перелічених алгоритмів: від базової ідеї та коректності до повного псевдокоду, варіантів

оптимізацій і типових сценаріїв застосування. Ми прагнутимемо не просто показати «як працює», а й пояснити «чому саме так» і «коли це найкращий вибір» для ваших даних.

10.4. 1. Перетворення Берроуза-Вілера

Теоретичні відомості

Перетворення Берроуза-Вілера (Burrows-Wheeler Transform, BWT) - це алгоритм перестановки символів у рядку, розроблений Майклом Берроузом та Девідом Вілером у 1994 році. Це перетворення не змінює символи рядка, а лише переставляє їх у спеціальний порядок, що робить результат більш придатним для стиснення. Основна властивість BWT полягає в тому, що воно групує однакові символи разом, зберігаючи при цьому можливість точного відновлення початкового рядка.

Алгоритм став основою для багатьох сучасних архіваторів, включаючи bzip2, і широко використовується в біоінформатиці для стиснення та індексації геномних даних. Унікальність перетворення полягає в його оборотності - з результату BWT можна однозначно відновити початковий рядок, знаючи лише позицію початкового символу в перетвореному рядку.

Основні концепції

Циклічні зсуви

Циклічний зсув рядка - це операція переміщення першого символу в кінець рядка. Для рядка S довжиною n існує рівно n різних циклічних зсувів:

- Зсув 0: початковий рядок S
- Зсув 1: S[1..n-1] + S[0]
- Зсув 2: S[2..n-1] + S[0..1]
- І так далі

Наприклад, для рядка “banana” циклічні зсуви:

- banana
- ananab
- nanaba
- anaban
- nabana

- abanan

Матриця циклічних зсувів

Матриця циклічних зсувів M - це матриця розміром n×n, де кожен рядок є циклічним зсувом початкового рядка. Рядки цієї матриці потім сортуються лексикографічно.

Алгоритм прямого перетворення

```
Алгоритм BWT_Transform(S):
    Вхід:
        S - вхідний рядок довжиною n
    Вихід:
        L - перетворений рядок (останній стовпець матриці)
        I - індекс початкового рядка в відсортованій матриці

    n ← довжина(S)

    // Створюємо всі циклічні зсуви
    rotations ← порожній список
    для i від 0 до n-1:
        rotation ← S[i..n-1] + S[0..i-1]
        додати rotation до rotations

    // Сортуємо циклічні зсуви лексикографічно
    відсортувати rotations

    // Знаходимо індекс початкового рядка
    I ← -1
    для i від 0 до n-1:
        якщо rotations[i] = S:
            I ← i
            вийти з циклу

    // Формуємо результат - останній стовпець
    L ← порожній рядок
    для i від 0 до n-1:
        додати rotations[i][n-1] до L

    повернути (L, I)
```

Оптимізований алгоритм з використанням суфіксного масиву

Наївний алгоритм має часову складність $O(n^2 \log n)$ через сортування рядків. Використовуючи суфіксний масив, можна оптимізувати до $O(n \log n)$ або навіть $O(n)$.

```
Алгоритм BWT_OptimizedTransform(S):
    Вхід:
        S - вхідний рядок довжиною n
    Вихід:
        L - перетворений рядок
        I - індекс початкового рядка

    n ← довжина(S)

    // Додаємо спеціальний символ для маркування кінця
    S_extended ← S + "$"

    // Будуємо суфіксний масив для подвоєного рядка
    S_doubled ← S + S
    SA ← побудувати_суфіксний_масив(S_doubled)

    // Фільтруємо суфікси, що відповідають циклічним зсувам
    rotations ← порожній список
    для i від 0 до довжина(SA)-1:
        якщо SA[i] < n:
            додати SA[i] до rotations

    // Формуємо BWT
    L ← порожній рядок
    I ← -1
    для i від 0 до n-1:
        pos ← rotations[i]
        якщо pos = 0:
            I ← i
            додати S[n-1] до L
        інакше:
            додати S[pos-1] до L

    повернути (L, I)
```

Алгоритм зворотного перетворення

Зворотне перетворення відновлює початковий рядок з BWT. Ключова ідея базується на властивості “First-Last”: i-те входження символу c в першому стовпці відсортованої матриці відповідає i-му входженню c в останньому стовпці.

```
Алгоритм BWT_Inverse(L, I):
    Вхід:
        L - перетворений рядок (останній стовпець)
        I - індекс початкового рядка
```

```
    Вихід:
        S - відновлений початковий рядок

    n ← довжина(L)

    // Крок 1: Обчислюємо перший стовпець
    F ← відсортована копія L

    // Крок 2: Будуємо таблицю відповідностей
    // Для кожної позиції в L знаходимо відповідну позицію в F
    T ← новий масив[n]

    // Підраховуємо кількість кожного символу
    count ← новий словник
    для c в L:
        якщо c не в count:
            count[c] ← 0
        count[c] ← count[c] + 1

    // Обчислюємо початкові позиції для кожного символу в F
    pos ← новий словник
    cumsum ← 0
    для c в відсортовані_ключі(count):
        pos[c] ← cumsum
        cumsum ← cumsum + count[c]

    // Будуємо таблицю відповідностей
    next_pos ← копія pos
    для i від 0 до n-1:
        c ← L[i]
        T[i] ← next_pos[c]
        next_pos[c] ← next_pos[c] + 1

    // Крок 3: Відновлюємо початковий рядок
    S ← порожній рядок
    j ← I
    для i від 0 до n-1:
        S ← F[j] + S
        j ← T[j]

    повернути S
```

Розглянемо детальний приклад перетворення для рядка “banana”:

Пряме перетворення:

1. Створюємо всі циклічні зсуви:

```
0: banana
1: ananab
2: nanaba
3: anaban
4: nabana
5: abanan
```
2. Сортуємо лексикографічно:

```
0: abanan
1: anaban
2: ananab
3: banana
4: nabana
5: nanaba
```
3. Беремо останній стовпець: “nnbaaa”

4. Індекс початкового рядка: I = 3

Зворотне перетворення:

1. L = “nnbaaa”, I = 3

2. F = відсортоване L = “aaabnn”

3. Будуємо таблицю відповідностей T = [3, 4, 2, 0, 1, 5]

4. Відновлюємо рядок, починаючи з позиції 3

Властивості перетворення

Групування символів

BWT має властивість групувати однакові символи поруч. Це відбувається тому, що рядки з однаковими суфіксами опиняються поруч після сортування, а їх останні символи (які передують цим суфіксам) часто однакові.

Оборотність

Перетворення є повністю оборотним - з пари (L, I) можна однозначно відновити початковий рядок S. Це досягається завдяки властивості First-Last.

Стійкість до локальних змін

Локальні зміни в початковому рядку призводять до локальних змін у перетвореному рядку, що робить BWT корисним для диференціального стиснення.

Розширені алгоритми

BWT для великих алфавітів

```
Алгоритм BWT_LargeAlphabet(S):
    Вхід:
        S - рядок з великим алфавітом
    Вихід:
        L - перетворений рядок
        I - індекс

    // Використовуємо радікс-сортування для ефективності
    n ← довжина(S)
    indices ← [0, 1, ..., n-1]

    // Сортуємо за допомогою ключових функцій
    функція key_at_offset(i, offset):
        повернути S[(i + offset) mod n]

    // Радікс-сортування по всіх позиціях
    для offset від n-1 до 0 з кроком -1:
        стабільно_відсортувати indices за key_at_offset(i, offset)

    // Формуємо результат
    L ← порожній рядок
    I ← -1
    для i від 0 до n-1:
        idx ← indices[i]
        якщо idx = 0:
            I ← i
        L ← L + S[(idx - 1 + n) mod n]

    повернути (L, I)
```

Потокове BWT

```
Алгоритм StreamingBWT(stream, block_size):
    Вхід:
        stream - вхідний потік даних
        block_size - розмір блоку для обробки
    Вихід:
        потік перетворених блоків

    buffer ← порожній рядок

    поки не кінець stream:
        // Читаємо дані в буфер
        поки довжина(buffer) < block_size та не кінець stream:
            buffer ← buffer + читати_символ(stream)
```

```
        якщо довжина(buffer) > 0:
            // Обробляємо блок
            (L, I) ← BWT_Transform(buffer)

            // Виводимо результат
            вивести I
            вивести L

            buffer ← порожній рядок
```

Інтеграція з алгоритмами стиснення

Move-to-Front кодування

Після BWT часто застосовується Move-to-Front (MTF) кодування для подальшого покращення стиснення:

```
Алгоритм MTF_Encode(S):
    Вхід:
        S - рядок після BWT
    Вихід:
        encoded - закодований рядок

    // Ініціалізуємо список символів
    alphabet ← унікальні_символи(S)
    відсортувати alphabet
    mtf_list ← список(alphabet)

    encoded ← порожній список
    для c в S:
        // Знаходимо позицію символу
        pos ← позиція c в mtf_list
        додати pos до encoded

        // Переміщуємо символ на початок
        видалити c з mtf_list на позиції pos
        вставити c в mtf_list на позицію 0

    повернути encoded
```

Run-Length кодування

Для послідовностей нулів після MTF ефективне Run-Length кодування:

```
Алгоритм RLE_Encode(data):
    Вхід:
        data - список чисел після MTF
    Вихід:
```

```
        encoded - закодовані дані

    encoded ← порожній список
    i ← 0

    поки i < довжина(data):
        якщо data[i] = 0:
            // Рахуємо послідовні нулі
            count ← 0
            поки i < довжина(data) та data[i] = 0:
                count ← count + 1
                i ← i + 1
            додати (0, count) до encoded
        інакше:
            додати data[i] до encoded
            i ← i + 1

    повернути encoded
```

Область застосування

Перетворення Берроуза-Вілера широко використовується в:

- Стиснення даних - основа архіваторів bzip2, використовується в 7-zip
- Біоінформатика - стиснення та індексація геномних послідовностей
- Обробка тексту - попередня обробка для ефективного пошуку
- Передача даних - зменшення обсягу даних для передачі
- Архівування - довготермінове зберігання великих обсягів даних
- Індексація - створення стиснених індексів для пошуку (FM-індекс)
- Обробка зображень - стиснення растрових зображень
- Криптографія - як етап попередньої обробки даних

Приклади застосування

Приклад 1: Стиснення тексту

Для тексту “the_theme_of_the_game” BWT дає “hgmfeeht_e__eetoha_m”, де однакові символи згруповані, що покращує стиснення.

Приклад 2: Стиснення ДНК послідовностей

ДНК послідовність “ACGTACGTACGT” після BWT стає “TTTTGGGCCCAAA”, що значно краще стискається.

Приклад 3: FM-індекс для пошуку

BWT є основою FM-індексу, який дозволяє виконувати пошук підрядків у стисненому тексті без розпакування.

Переваги та недоліки

Переваги

- Покращення стиснення - групування однакових символів підвищує ефективність подальшого стиснення

- Повна оборотність - можливість точного відновлення початкових даних

- Локальність - зберігає локальні властивості тексту

- Універсальність - працює з будь-якими типами даних

- Паралелізація - можливість обробки блоками

- Основа для індексів - використовується в сучасних стиснених індексах

Недоліки

- Використання пам’яті - потребує O(n) додаткової пам’яті для обробки

- Складність реалізації - ефективні алгоритми досить складні

- Час обробки - повільніше за прості методи стиснення

- Блочність - необхідність розбиття великих файлів на блоки

- Чутливість до типу даних - не всі дані добре стискаються

- Накладні витрати - для малих файлів накладні витрати можуть перевищувати вигоду

Контрольні запитання

1. Що таке перетворення Берроуза-Вілера і яка його основна мета?

2. Як створюється матриця циклічних зсувів?

3. Що таке властивість First-Last і як вона використовується при зворотному перетворенні?

4. Чому BWT покращує стиснення даних?

5. Яка часова складність наївного алгоритму BWT?

6. Як можна оптимізувати BWT використовуючи суфіксний масив?

7. Що таке Move-to-Front кодування і чому воно ефективне після BWT?

8. Як працює зворотне перетворення BWT?

9. Для яких типів даних BWT найбільш ефективне?

10. Як BWT використовується в сучасних архіваторах?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження перетворення Берроуза-Вілера

2. Мета

Вивчити принципи роботи перетворення Берроуза-Вілера, реалізувати алгоритми прямого та зворотного перетворення, дослідити вплив на ефективність стиснення даних.

3. Завдання

- Реалізувати наївний алгоритм BWT
- Реалізувати оптимізований алгоритм з суфіксним масивом
- Створити алгоритм зворотного перетворення
- Реалізувати MTF кодування
- Інтегрувати з простим алгоритмом стиснення
- Візуалізувати процес перетворення
- Дослідити ефективність для різних типів даних
- Реалізувати блочну обробку великих файлів

4. Опис

Лабораторна робота охоплює повний цикл роботи з перетворенням Берроуза-Вілера: від базової реалізації до інтеграції з алгоритмами стиснення. Студенти мають зрозуміти математичні основи перетворення та дослідити його практичне застосування.

5. Порядок виконання

- Створити клас BurrowsWheeler з основними методами
- Реалізувати метод transform для прямого перетворення
- Імплементувати метод inverseTransform
- Додати метод optimizedTransform з суфіксним масивом
- Створити клас MTFEncoder
- Реалізувати візуалізатор BWTVisualizer
- Інтегрувати з алгоритмом стиснення
- Провести тестування на різних даних
- Виконати аналіз ефективності

6. Результат

Програма повинна: - Виконувати пряме та зворотне перетворення BWT - Демонструвати групування символів - Показувати покращення стиснення - Візуалізувати етапи перетворення - Обробляти файли різних типів - Надавати статистику ефективності

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас BurrowsWheeler з методами перетворення

- Метод transform з детальними коментарями

- Метод inverseTransform з поясненням логіки

- Клас MTFEncoder для Move-to-Front кодування

- Модуль візуалізації процесу

- Модуль інтеграції зі стисненням

2. Продемонструвати роботу додатку:

- Перетворення коротких рядків з візуалізацією

- Зворотне перетворення з перевіркою коректності

- Групування символів після BWT

- Ефект MTF кодування

- Порівняння розмірів до та після стиснення

- Обробка різних типів файлів

3. Бути готовим відповісти на питання:

- Як працює властивість First-Last?

- Чому BWT покращує стиснення?

- Як вибрати оптимальний розмір блоку?

- Яка роль MTF кодування?

- Як оптимізувати алгоритм для великих файлів?

Тести

1. Для рядка “banana”, який символ буде першим у результаті BWT?

1) b

2) a

3) n

4) $

2. Яка часова складність наївного алгоритму BWT для рядка довжини n?

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) $O(n^2 \log n)$

3. Що зберігається разом з перетвореним рядком для можливості зворотного перетворення?

1) Початковий рядок

2) Індекс початкового рядка в матриці

3) Всі циклічні зсуви

4) Суфіксний масив

4. Яка основна властивість BWT, що покращує стиснення?

1) Зменшення ентропії

2) Групування однакових символів

3) Видалення повторів

4) Кодування довжин серій

5. Move-to-Front кодування після BWT ефективне, тому що:

1) Видаляє дублікати

2) Створює багато малих чисел

3) Збільшує ентропію

4) Зменшує розмір алфавіту

Відповіді:

1. c)

2. d)

3. b)

4. b)

5. b)

Тести підвищеної складності

1. Скільки різних результатів BWT може дати рядок з n різних символів?

1) 1

2) n

3) n!

4) n!/n = (n-1)!

2. Яка просторова складність оптимального алгоритму зворотного BWT?

1) O(1)

2) O(n)

3) O(n log n)

4) $O(n^2)$

3. Для рядка з k різних символів, яка максимальна довжина серії в MTF після BWT?

1) n

2) n/k

3) k

4) log n

4. При використанні суфіксного масиву для BWT, яка операція найскладніша?

1) Побудова суфіксного масиву

2) Фільтрація циклічних зсувів

3) Формування результату

4) Знаходження індексу

5. Який розмір блоку оптимальний для BWT в bzip2?

1) 100 КБ

2) 900 КБ

3) 1 МБ

4) 10 МБ

6. Скільки проходів по даних потрібно для зворотного BWT?

1) 1

2) 2

3) log n

4) n

7. Яка структура даних найефективніша для реалізації MTF?

1) Масив

2) Зв’язаний список

3) Дерево

4) Хеш-таблиця

8. BWT зберігає яку властивість початкового тексту?

1) Порядок слів

2) k-грами при циклічному читанні

3) Частоти символів

4) Всі перелічені

9. Для якого типу даних BWT найменш ефективне?

1) Природна мова

2) ДНК послідовності

3) Випадкові дані

4) Програмний код

10. FM-індекс використовує BWT для:

1) Стиснення

2) Пошуку без розпакування

3) Шифрування

4) Хешування

Відповіді:

1. d) (n-1)! - кожна циклічна перестановка дає унікальний результат

2. b) O(n) - потрібен масив для таблиці відповідностей

3. a) n - в гіршому випадку всі символи однакові

4. a) - Побудова суфіксного масиву домінує в складності

5. b) 900 КБ - стандартний розмір блоку в bzip2

6. a) 1 - достатньо одного проходу з правильною структурою даних

7. b) - Зв’язаний список ефективний для операції move-to-front

8. d) - BWT зберігає всі перелічені властивості

9. c) - Випадкові дані не мають структури для групування

10. b) - FM-індекс дозволяє пошук у стисненому тексті

Завдання для самостійної роботи

1. Реалізуйте BWT для алфавіту Unicode з ефективною обробкою багатобайтових символів.

2. Створіть паралельну версію BWT для багатоядерних процесорів з розподілом блоків.

3. Розробіть адаптивний алгоритм вибору розміру блоку залежно від типу даних.

4. Імплементуйте двонаправлене BWT, яке можна інвертувати з обох кінців.

5. Створіть інкрементальний алгоритм BWT для потокових даних.

6. Реалізуйте оптимізацію BWT для повторюваних патернів використовуючи граматичне стиснення.

7. Розробіть гібридний алгоритм, що комбінує BWT з іншими перетвореннями.

8. Створіть візуальний аналізатор для дослідження впливу BWT на різні типи даних.

9. Імплементуйте BWT-based алгоритм для пошуку приблизних збігів.

10. Розробіть систему автоматичного налаштування параметрів BWT для оптимального стиснення.

Теми лабораторних робіт

1. Розробка архіватора на основі BWT з підтримкою різних алгоритмів вторинного стиснення.

2. Створення системи стиснення геномних даних з використанням спеціалізованого BWT.

3. Імплементація FM-індексу для ефективного пошуку в стиснених текстах.

4. Розробка інструменту порівняння ефективності BWT для різних мов та алфавітів.

5. Створення системи передачі даних з BWT-компресією в реальному часі.

6. Реалізація бібліотеки стиснення зображень на основі двовимірного BWT.

7. Розробка криптографічної системи з використанням BWT як етапу обфускації.

8. Створення адаптивного компресора, що вибирає між BWT та іншими методами.

9. Імплементація розподіленої системи обробки великих даних з BWT.

10. Розробка навчального середовища для візуалізації та експериментів з BWT.

10.4.2 Алгоритми стиснення LZ77 та LZ78

Теоретичні відомості

Алгоритми LZ77 та LZ78 були розроблені Абрахамом Лемпелем та Якобом Зівом у 1977 та 1978 роках відповідно. Ці алгоритми стали основою для цілого сімейства методів стиснення даних без втрат, які використовуються в багатьох сучасних архіваторах та форматах файлів. Основна ідея обох алгоритмів полягає в заміні повторюваних послідовностей символів посиланнями на їх попередні входження, що дозволяє досягти значного ступеня стиснення для даних з повторюваними патернами.

LZ77 та LZ78 представляють два різні підходи до словникового стиснення. LZ77 використовує ковзне вікно як неявний словник, тоді як LZ78 будує явний словник фраз під час процесу стиснення. Обидва алгоритми є адаптивними - вони не потребують попереднього знання про статистичні властивості даних і пристосовуються до них під час роботи.

Алгоритм LZ77

Основні концепції

LZ77 базується на концепції ковзного вікна, яке складається з двох частин: - Буфер пошуку (search buffer) - містить вже оброблені символи - Буфер попереднього перегляду (lookahead buffer) - містить символи, які потрібно закодувати

Алгоритм шукає найдовше співпадіння початку буфера попереднього перегляду з будь-якою підпослідовністю в буфері пошуку.

Структура коду LZ77

Вихід алгоритму LZ77 складається з послідовності трійок (offset, length, next_symbol), де: - offset - відстань від поточної позиції до початку

знайденого співпадіння - length - довжина співпадіння - next_symbol - символ, що йде після співпадіння

Алгоритм стиснення LZ77

```
Алгоритм LZ77_Compress(input):
    Вхід:
        input - вхідний рядок для стиснення
    Вихід:
        compressed - список трійок (offset, length, next_symbol)

    n ← довжина(input)
    search_buffer_size ← 32768  // типовий розмір
    lookahead_buffer_size ← 258  // типовий розмір

    compressed ← порожній список
    position ← 0

    поки position < n:
        // Визначаємо межі буферів
        search_start ← max(0, position - search_buffer_size)
        search_end ← position
        lookahead_end ← min(position + lookahead_buffer_size, n)

        // Знаходимо найдовше співпадіння
        best_offset ← 0
        best_length ← 0

        для i від search_start до search_end - 1:
            // Порівнюємо з поточною позицією
            match_length ← 0
            поки (i + match_length < search_end та
                  position + match_length < lookahead_end та
                  input[i + match_length] = input[position + match_l
ength]):
                match_length ← match_length + 1

            // Оновлюємо найкраще співпадіння
            якщо match_length > best_length:
                best_offset ← position - i
                best_length ← match_length

        // Формуємо вихідну трійку
        якщо position + best_length < n:
            next_char ← input[position + best_length]
        інакше:
```

```
            next_char ← EOF  // спеціальний символ кінця

        додати (best_offset, best_length, next_char) до compressed

        // Просуваємось вперед
        position ← position + best_length + 1

    повернути compressed
```

Алгоритм декомпресії LZ77

```
Алгоритм LZ77_Decompress(compressed):
    Вхід:
        compressed - список трійок (offset, length, next_symbol)
    Вихід:
        output - відновлений рядок

    output ← порожній рядок

    для кожної трійки (offset, length, next_symbol) в compressed:
        якщо length > 0:
            // Копіюємо символи з попередньої позиції
            start_pos ← довжина(output) - offset
            для i від 0 до length - 1:
                // Обробка випадку, коли копіюємо з області, що перек
ривається
                char_to_copy ← output[start_pos + i]
                додати char_to_copy до output

        // Додаємо наступний символ
        якщо next_symbol ≠ EOF:
            додати next_symbol до output

    повернути output
```

Оптимізації пошуку в LZ77

```
Алгоритм LZ77_OptimizedSearch(input, position, search_buffer_size, lo
okahead_buffer_size):
    Вхід:
        input - вхідний рядок
        position - поточна позиція
        search_buffer_size - розмір буфера пошуку
        lookahead_buffer_size - розмір буфера перегляду
    Вихід:
        (offset, length) - найкраще співпадіння

    // Використовуємо хеш-таблицю для швидкого пошуку
```

```
    hash_table ← новий словник
    search_start ← max(0, position - search_buffer_size)

    // Будуємо хеш-таблицю для трійок символів
    для i від search_start до position - 3:
        якщо i + 2 < position:
            key ← (input[i], input[i+1], input[i+2])
            якщо key не в hash_table:
                hash_table[key] ← порожній список
            додати i до hash_table[key]

    // Шукаємо співпадіння
    best_offset ← 0
    best_length ← 0

    якщо position + 2 < довжина(input):
        search_key ← (input[position], input[position+1], input[posit
ion+2])
        якщо search_key в hash_table:
            для start_pos в hash_table[search_key]:
                // Розширюємо співпадіння
                length ← 3
                max_length ← min(position - start_pos, lookahead_buff
er_size)

                поки (length < max_length та
                      start_pos + length < position та
                      position + length < довжина(input) та
                      input[start_pos + length] = input[position + l
ength]):
                    length ← length + 1

                якщо length > best_length:
                    best_offset ← position - start_pos
                    best_length ← length

    повернути (best_offset, best_length)
```

Алгоритм LZ78

Основні концепції

На відміну від LZ77, алгоритм LZ78 будує явний словник фраз під час стиснення. Кожна нова фраза складається з існуючої фрази зі словника плюс один новий символ. Словник починається порожнім і поступово заповнюється унікальними фразами.

Структура коду LZ78

Вихід алгоритму LZ78 складається з послідовності пар (index, symbol), де: - index - індекс фрази в словнику (0 означає порожню фразу) - symbol - новий символ, що додається до фрази

Алгоритм стиснення LZ78

```
Алгоритм LZ78_Compress(input):
    Вхід:
        input - вхідний рядок для стиснення
    Вихід:
        compressed - список пар (index, symbol)

    dictionary ← новий словник
    dictionary_size ← 0
    compressed ← порожній список

    current_phrase ← порожній рядок
    phrase_index ← 0

    для кожного символу char в input:
        current_phrase ← current_phrase + char

        якщо current_phrase в dictionary:
            // Фраза вже є в словнику, продовжуємо
            phrase_index ← dictionary[current_phrase]
        інакше:
            // Нова фраза, додаємо до виходу та словника
            додати (phrase_index, char) до compressed

            // Додаємо нову фразу до словника
            dictionary_size ← dictionary_size + 1
            dictionary[current_phrase] ← dictionary_size

            // Починаємо нову фразу
            current_phrase ← порожній рядок
            phrase_index ← 0

    // Обробка залишку
    якщо current_phrase ≠ порожній рядок:
        last_char ← останній символ current_phrase
        додати (phrase_index, last_char) до compressed

    повернути compressed
```

Алгоритм декомпресії LZ78

```
Алгоритм LZ78_Decompress(compressed):
    Вхід:
        compressed - список пар (index, symbol)
    Вихід:
        output - відновлений рядок

    dictionary ← новий масив
    dictionary[0] ← порожній рядок  // індекс 0 - порожня фраза
    dictionary_size ← 0

    output ← порожній рядок

    для кожної пари (index, symbol) в compressed:
        // Відновлюємо фразу
        якщо index = 0:
            phrase ← порожній рядок
        інакше:
            phrase ← dictionary[index]

        // Додаємо новий символ
        phrase ← phrase + symbol

        // Виводимо фразу
        додати phrase до output

        // Додаємо до словника
        dictionary_size ← dictionary_size + 1
        dictionary[dictionary_size] ← phrase

    повернути output
```

Модифікації та покращення

1. LZW (Lempel-Ziv-Welch)

LZW є модифікацією LZ78, розробленою Террі Велчем. Основна відмінність - ініціалізація словника всіма можливими однобайтовими символами.

```
Алгоритм LZW_Compress(input):
    Вхід:
        input - вхідний рядок
    Вихід:
        compressed - список кодів

    // Ініціалізуємо словник всіма символами
    dictionary ← новий словник
    для і від 0 до 255:
```

```
        dictionary[символ(i)] ← i

    next_code ← 256
    compressed ← порожній список

    current ← порожній рядок

    для кожного символу char в input:
        combined ← current + char

        якщо combined в dictionary:
            current ← combined
        інакше:
            // Виводимо код поточної фрази
            додати dictionary[current] до compressed

            // Додаємо нову фразу до словника
            якщо next_code < MAX_DICT_SIZE:
                dictionary[combined] ← next_code
                next_code ← next_code + 1

            current ← char

    // Виводимо останню фразу
    якщо current ≠ порожній рядок:
        додати dictionary[current] до compressed

    повернути compressed
```

2. LZSS (Lempel-Ziv-Storer-Szymanski)

LZSS є модифікацією LZ77, яка усуває надлишковість кодування окремих символів.

```
Алгоритм LZSS_Compress(input):
    Вхід:
        input - вхідний рядок
    Вихід:
        compressed - бітовий потік

    min_match_length ← 3  // мінімальна довжина для кодування посилан
ням

    compressed ← новий бітовий_потік
    position ← 0

    поки position < довжина(input):
```

```
        (offset, length) ← знайти_найкраще_співпадіння(input, positio
n)

        якщо length >= min_match_length:
            // Кодуємо як посилання
            записати_біт(compressed, 1)  // флаг посилання
            записати_число(compressed, offset, біти_для_offset)
            записати_число(compressed, length, біти_для_довжини)
            position ← position + length
        інакше:
            // Кодуємо як літерал
            записати_біт(compressed, 0)  // флаг літерала
            записати_символ(compressed, input[position])
            position ← position + 1

    повернути compressed
```

Порівняння LZ77 та LZ78

Використання пам’яті

LZ77 використовує фіксований об’єм пам’яті для ковзного вікна, тоді як LZ78 потребує пам’яті пропорційно кількості унікальних фраз у вхідних даних.

Швидкість стиснення

LZ77 може бути повільнішим через необхідність пошуку в буфері, але з хешуванням може досягти швидкості, порівнянної з LZ78.

Ступінь стиснення

LZ77 зазвичай досягає кращого стиснення для даних з довгими повторюваними послідовностями, тоді як LZ78 ефективніший для даних з короткими повторюваними патернами.

Практичні реалізації

Структури даних для оптимізації

```
Структура SuffixTree:
    // Для швидкого пошуку в LZ77
    root - корінь дерева

    функція insert(position):
        // Вставка нового суфікса

    функція find_longest_match(pattern):
        // Пошук найдовшого співпадіння
```

```
Структура TrieNode:
    // Для ефективного словника в LZ78
    children - словник дочірніх вузлів
    phrase_index - індекс фрази

    функція insert(phrase, index):
        // Вставка нової фрази

    функція search(phrase):
        // Пошук фрази в трі
```

Область застосування

Алгоритми LZ77 та LZ78 та їх модифікації широко використовуються в:

- Архіватори - ZIP (deflate базується на LZ77), GIF та TIFF (LZW)

- Системи передачі даних - модеми V.42bis використовують LZW

- Файлові системи - NTFS підтримує прозоре стиснення на основі LZ

- Мережеві протоколи - HTTP стиснення (gzip, deflate)

- Бази даних - стиснення сторінок та резервних копій

- Вбудовані системи - стиснення прошивок та даних

- Медичні зображення - стиснення без втрат для DICOM

- Текстові процесори - внутрішнє представлення документів

Приклади застосування

Приклад 1: Стиснення тексту з повтореннями

Для тексту “the_rain_in_spain_falls_mainly_in_the_plain” LZ77 ефективно кодує повторення “in_”, “*the*”, “ain” використовуючи посилання назад.

Приклад 2: Стиснення програмного коду

Програмний код часто містить повторювані патерни (ключові слова, ідентифікатори), що робить LZ-алгоритми ефективними для його стиснення.

Приклад 3: Стиснення логів

Лог-файли з повторюваними записами досягають високого ступеня стиснення завдяки можливості посилатися на попередні схожі записи.

Переваги та недоліки

Переваги LZ77

- Обмежене використання пам’яті - фіксований розмір вікна

- Простота реалізації - концептуально простий алгоритм

- Добре стискає довгі повтори - ефективний для текстів з довгими повторюваними фрагментами

- Не потребує передачі словника - словник неявний

- Стійкість до помилок - локальні пошкодження не поширюються

Недоліки LZ77

- Повільний пошук - без оптимізацій пошук може бути повільним

- Обмежена дальність посилань - розмір вікна обмежує можливості

- Надлишковість для окремих символів - кодує навіть окремі символи трійками

Переваги LZ78

- Швидкий пошук - використання словника прискорює пошук

- Адаптивність - автоматично адаптується до даних

- Ефективний для коротких патернів - добре працює з короткими повтореннями

- Простота декодування - декомпресія дуже швидка

Недоліки LZ78

- Необмежене зростання словника - може потребувати багато пам’яті

- Повільний старт - на початку словник порожній

- Необхідність очищення словника - для довгих файлів потрібно періодично очищати

- Менша ефективність для довгих повторів - не може ефективно кодувати дуже довгі повтори

Контрольні запитання

1. В чому основна відмінність між LZ77 та LZ78?

2. Що таке ковзне вікно в алгоритмі LZ77?

3. Як працює словник в алгоритмі LZ78?

4. Яка структура виходу алгоритму LZ77?

5. Чому LZW ініціалізує словник всіма можливими символами?

6. Як LZSS покращує LZ77?

7. Які фактори впливають на ефективність стиснення LZ77?

8. Коли словник LZ78 потребує очищення?

9. Яка часова складність пошуку в LZ77 без оптимізацій?

10. Для яких типів даних LZ-алгоритми найбільш ефективні?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритмів стиснення LZ77 та LZ78

2. Мета

Вивчити принципи словникового стиснення, реалізувати алгоритми LZ77 та LZ78, дослідити їх ефективність для різних типів даних.

3. Завдання

- Реалізувати базовий алгоритм LZ77 з пошуком у вікні

- Оптимізувати пошук використовуючи хешування

- Реалізувати алгоритм LZ78 з динамічним словником

- Створити варіант LZW

- Реалізувати декомпресори для всіх алгоритмів

- Візуалізувати процес стиснення

- Порівняти ефективність алгоритмів

- Дослідити вплив параметрів на стиснення

4. Опис

Лабораторна робота охоплює реалізацію двох фундаментальних алгоритмів словникового стиснення. Студенти мають зрозуміти різницю між підходами з ковзним вікном та явним словником, реалізувати ефективні структури даних для пошуку та дослідити практичні аспекти стиснення.

5. Порядок виконання

- Створити клас LZ77Compressor з методами compress та decompress

- Реалізувати наївний пошук у вікні

- Додати оптимізацію з хеш-таблицею

- Створити клас LZ78Compressor

- Імплементувати трі-структуру для словника

- Реалізувати LZWCompressor

- Створити візуалізатор процесу стиснення

- Провести тестування на різних даних

- Побудувати графіки залежності ступеня стиснення від параметрів

6. Результат

Програма повинна: - Стискати та розпаковувати файли різних типів - Показувати процес роботи алгоритмів покроково - Відображати статистику стиснення - Візуалізувати вміст словника/вікна - Порівнювати швидкість та ступінь стиснення - Зберігати стиснені дані у власному форматі

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас LZ77Compressor з оптимізованим пошуком

- Клас LZ78Compressor з трі-словником

- Клас LZWCompressor

- Модулі декомпресії для всіх алгоритмів

- Візуалізатор процесу стиснення

- Модуль порівняльного аналізу

2. Продемонструвати роботу додатку:

- Стиснення текстових файлів різних розмірів

- Візуалізація пошуку співпадінь у LZ77

- Показ росту словника в LZ78

- Порівняння ступенів стиснення

- Вплив розміру вікна на ефективність LZ77

- Обробка різних типів даних (текст, код, бінарні)

3. Бути готовим відповісти на питання:

- Як вибрати оптимальний розмір вікна для LZ77?

- Коли потрібно очищати словник в LZ78?

- Чому LZSS ефективніший за LZ77?

- Як оптимізувати пошук у ковзному вікні?

- Які переваги дає ініціалізація словника в LZW?

Тести

1. Яка структура виходу алгоритму LZ77?

1) Пари (індекс, символ)

2) Трійки (зсув, довжина, наступний символ)

3) Лише стиснені дані

4) Словник та стиснені дані

2. Що зберігає словник в алгоритмі LZ78?

1) Лише окремі символи

2) Фрази змінної довжини

3) Фіксовані блоки даних

4) Хеш-значення фраз

3. Яка основна перевага LZW над LZ78?

1) Менше використання пам’яті

2) Швидша компресія

3) Не потребує передачі початкових символів

4) Кращий ступінь стиснення

4. Типовий розмір буфера пошуку в LZ77:

1) 256 байт

2) 1 КБ

3) 32 КБ

4) 1 МБ

5. Коли словник LZ78 потребує очищення?

1) Ніколи

2) Після кожного символу

3) При досягненні максимального розміру

4) Після кожного блоку даних

Відповіді:

1. b) - LZ77 виводить трійки (offset, length, next_symbol)

2. b) - Словник зберігає фрази змінної довжини

3. c) - LZW ініціалізує словник всіма символами алфавіту

4. c) - Типовий розмір 32 КБ (32768 байт)

5. c) - Словник очищається при досягненні ліміту

Тести підвищеної складності

1. Яка часова складність наївного пошуку в LZ77 для тексту довжини n з вікном розміру w?

1) O(n)

2) O(n × w)

3) O(n × w^2)

4) O(n^2)

2. Максимальна кількість фраз у словнику LZ78 для алфавіту розміру σ після обробки n символів:

1) n

2) σ^n

3) O(√n)

4) O(n/log n)

3. При якому типі даних LZ77 буде ефективніший за LZ78?

1) Випадкові дані

2) Текст з довгими повтореннями

3) Короткі повторювані патерни

4) Бінарні дані

4. Яка структура даних найефективніша для пошуку в LZ77?

1) Лінійний масив

2) Хеш-таблиця

3) Суфіксне дерево

4) B-дерево

5. Скільки біт потрібно для кодування зсуву в LZ77 з вікном 32КБ?

1) 8

2) 12

3) 15

4) 16

6. Чому в LZSS використовується мінімальна довжина співпадіння?

1) Для швидшої компресії

2) Щоб уникнути кодування коротких співпадінь

3) Для зменшення словника

4) Для кращої декомпресії

7. Яка максимальна довжина фрази в LZ78 після обробки n символів?

1) log n

2) $\sqrt{n}$

3) n/2

4) n

8. При стисненні якого файлу LZ-алгоритми будуть найменш ефективні?

1) Текстовий документ

2) Програмний код

3) Зашифровані дані

4) XML файл

9. Скільки пам’яті потребує декомпресор LZ77?

1) O(1)

2) O(розмір вікна)

3) O(розмір файлу)

4) O(розмір словника)

10. Яке співвідношення між LZ77 та алгоритмом deflate?

1) Deflate = LZ77

2) Deflate = LZ77 + Huffman

3) Deflate = LZ78 + Huffman

4) Deflate = LZW + Huffman

Відповіді:

1. c) $O(n \times w^2)$ - для кожної позиції перевіряємо w позицій з довжиною до w

2. c) $O(\sqrt{n})$ - теоретична оцінка для оптимального розбиття

3. b) - LZ77 ефективніший для довгих повторюваних послідовностей

4. c) - Суфіксне дерево дає найшвидший пошук O(m)

5. c) 15 біт - для адресації 32768 байт

6. b) - Короткі співпадіння неефективно кодувати трійками

7. a) log n - середня довжина фрази в оптимальному розбитті

8. c) - Зашифровані дані виглядають випадковими без повторень

9. b) O(розмір вікна) - потрібно зберігати лише вікно декомпресованих даних

10. b) - Deflate використовує LZ77 з подальшим кодуванням Хаффмана

Завдання для самостійної роботи

1. Реалізуйте адаптивну версію LZ77, яка динамічно змінює розмір вікна залежно від характеристик даних.

2. Створіть паралельну версію LZ77 з розбиттям на блоки та незалежною обробкою.

3. Розробіть гібридний алгоритм, що автоматично перемикається між LZ77 та LZ78 залежно від типу даних.

4. Імплементуйте LZ78 з ієрархічним словником для ефективної роботи з великими файлами.

5. Створіть модифікацію LZSS з змінною мінімальною довжиною співпадіння.

6. Реалізуйте потокову версію LZ77 для стиснення даних у реальному часі.

7. Розробіть алгоритм попереднього аналізу даних для вибору оптимальних параметрів LZ.

8. Створіть версію LZW з адаптивним розміром коду для кращого стиснення.

9. Імплементуйте алгоритм виявлення та усунення циклів у словнику LZ78.

10. Розробіть систему бенчмаркінгу для порівняння різних варіантів LZ-алгоритмів.

Теми лабораторних робіт

1. Розробка архіватора з підтримкою LZ77, LZ78 та LZW з автоматичним вибором алгоритму.

2. Створення системи прозорого стиснення файлів для операційної системи на основі LZ.

3. Імплементація HTTP-сервера з підтримкою стиснення контенту алгоритмами LZ.

4. Розробка бібліотеки стиснення для вбудованих систем з обмеженими ресурсами.

5. Створення інструменту аналізу ефективності LZ-алгоритмів для різних типів даних.

6. Реалізація системи диференціального бекапу з використанням модифікованого LZ77.

7. Розробка стиснутої файлової системи на основі блочного LZ-стиснення.

8. Створення мережевого протоколу з вбудованим LZ-стисненням для IoT пристроїв.

9. Імплементація GPU-прискореної версії LZ77 для обробки великих обсягів даних.

10. Розробка навчального середовища для візуалізації та експериментів з LZ-алгоритмами.

10.4. Алгоритм кодування Хаффмана

Теоретичні відомості

Алгоритм Хаффмана - це жадібний алгоритм побудови оптимального префіксного коду для стиснення даних без втрат. Основна ідея полягає в тому, що символам, які зустрічаються частіше, присвоюються коротші бітові коди, а рідкісним символам - довші коди. Це дозволяє мінімізувати середню довжину закодованого повідомлення.

Ключовий принцип алгоритму базується на спостереженні: якщо ми хочемо ефективно закодувати текст, то найчастіші символи повинні мати найкоротше представлення. Наприклад, у звичайному англійському тексті літера ‘e’ зустрічається набагато частіше за літеру ‘z’, тому логічно дати їй коротший код.

Алгоритм створює бінарне дерево, де:

- Кожен листок представляє символ алфавіту

- Шлях від кореня до листка визначає бінарний код символу

- Ліва гілка означає біт ‘0’, права - біт ‘1’

- Символи з більшою частотою розташовані ближче до кореня

Префіксність коду означає, що жоден код не є початком (префіксом) іншого коду. Це забезпечує однозначність декодування - ми можемо читати

закодований потік бітів і точно визначати, де закінчується один символ і починається інший.

Розглянемо детальніше процес побудови дерева Хаффмана. Процес побудови дерева Хаффмана складається з кількох чітко визначених етапів:

Етап 1: Ініціалізація

Створюється набір вузлів-листків, по одному для кожного символу алфавіту. Кожен вузол містить символ та його частоту (або ймовірність). Усі ці вузли поміщаються в пріоритетну чергу (мін-купу), де пріоритет визначається частотою вузла — вузли з меншими частотами мають вищий пріоритет для вилучення.

Етап 2: Побудова дерева

Основний цикл алгоритму виконується, поки в черзі залишається більше одного вузла:

- Вилучаються два вузли з найменшими частотами з черги

- Створюється новий внутрішній вузол, який стає батьком цих двох вузлів

- Частота нового вузла встановлюється як сума частот його нащадків

- Лівому нащадку присвоюється біт “0”, правому — біт “1”

- Новий вузол додається назад до черги

Етап 3: Формування кодів

Коли в черзі залишається лише один вузол, він стає коренем дерева. Коди для кожного символу формуються шляхом проходження від кореня до відповідного листка: - Рух ліворуч додає “0” до коду - Рух праворуч додає “1” до коду

Крок 1: Аналіз частот

Спочатку необхідно проаналізувати вхідні дані та підрахувати частоту кожного символу.

```
Алгоритм AnalyzeFrequencies(data):
    Вхід:
        data - масив символів для кодування
    Вихід:
        frequencyTable - таблиця частот символів

    frequencyTable ← новий словник
```

```
    // Проходимо по всіх символах
    для кожного символу symbol в data:
        якщо symbol існує в frequencyTable:
            frequencyTable[symbol] ← frequencyTable[symbol] + 1
        інакше:
            frequencyTable[symbol] ← 1

    повернути frequencyTable
```

Крок 2: Створення листкових вузлів

Для кожного унікального символу створюємо вузол дерева.

```
Структура HuffmanNode:
    symbol      // символ (null для внутрішніх вузлів)
    frequency   // частота або сумарна вага
    leftChild   // лівий нащадок
    rightChild  // правий нащадок

    // Конструктор для листкового вузла
    конструктор CreateLeaf(sym, freq):
        symbol ← sym
        frequency ← freq
        leftChild ← null
        rightChild ← null

    // Конструктор для внутрішнього вузла
    конструктор CreateInternal(left, right):
        symbol ← null
        frequency ← left.frequency + right.frequency
        leftChild ← left
        rightChild ← right

    // Перевірка чи вузол є листком
    функція isLeaf():
        повернути (leftChild = null та rightChild = null)
```

Крок 3: Ініціалізація пріоритетної черги

Використовуємо мін-купу (пріоритетну чергу), де вузли впорядковані за частотою.

```
Алгоритм InitializePriorityQueue(frequencyTable):
    Вхід:
        frequencyTable - таблиця частот
    Вихід:
        priorityQueue - пріоритетна черга вузлів

    priorityQueue ← нова МінКупа()
```

```
    // Створюємо листковий вузол для кожного символу
    для кожної пари (symbol, frequency) в frequencyTable:
        node ← новий HuffmanNode.CreateLeaf(symbol, frequency)
        priorityQueue.insert(node, frequency)

    повернути priorityQueue
```

Крок 4: Побудова дерева знизу вгору

Це серце алгоритму Хаффмана - ітеративне об’єднання вузлів з найменшими частотами.

```
Алгоритм BuildHuffmanTree(frequencyTable):
    Вхід:
        frequencyTable - таблиця частот символів
    Вихід:
        root - корінь дерева Хаффмана

    // Ініціалізація
    priorityQueue ← InitializePriorityQueue(frequencyTable)

    // Спеціальний випадок: лише один унікальний символ
    якщо priorityQueue.size() = 1:
        leaf ← priorityQueue.extractMin()
        root ← новий HuffmanNode.CreateInternal(leaf, null)
        повернути root

    // Основний цикл побудови дерева
    поки priorityQueue.size() > 1:
        // Крок 1: Витягуємо два вузли з найменшими частотами
        firstMin ← priorityQueue.extractMin()
        secondMin ← priorityQueue.extractMin()

        // Крок 2: Створюємо новий внутрішній вузол
        // Частота нового вузла = сума частот дочірніх
        newNode ← новий HuffmanNode.CreateInternal(firstMin, secondMi
n)

        // Крок 3: Вставляємо новий вузол назад у чергу
        priorityQueue.insert(newNode, newNode.frequency)

        // Черга зменшилась на 1 елемент

    // Останній вузол у черзі - це корінь дерева
    root ← priorityQueue.extractMin()
    повернути root
```

Розглянемо покроковий приклад для рядка “ABRACADABRA”.

Крок 1: Підрахунок частот

- A: 5 разів

- B: 2 рази

- R: 2 рази

- C: 1 раз

- D: 1 раз

Крок 2: Створення листкових вузлів та ініціалізація черги

Початковий стан черги (впорядкована за частотою):

```
[C:1] [D:1] [B:2] [R:2] [A:5]
```

Крок 3: Перша ітерація

- Витягуємо C:1 та D:1

- Створюємо внутрішній вузол з частотою 1+1=2

- Вставляємо назад у чергу

```
      (2)
     /   \
   C:1   D:1
```

```
Черга: [(CD):2] [B:2] [R:2] [A:5]
```

Крок 4: Друга ітерація

- Витягуємо (CD):2 та B:2

- Створюємо внутрішній вузол з частотою 2+2=4

```
       (4)
      /   \
    (2)   B:2
   /   \
 C:1   D:1
```

```
Черга: [R:2] [(CDB):4] [A:5]
```

Крок 5: Третя ітерація

- Витягуємо R:2 та (CDB):4

- Створюємо внутрішній вузол з частотою 2+4=6

```
         (6)
        /   \
      R:2   (4)
```

```
        /   \
      (2)   B:2
     /   \
  C:1   D:1
```

```
Черга: [A:5] [(RCDB):6]
```

Крок 6: Четверта ітерація

- Витягуємо A:5 та (RCDB):6

- Створюємо кореневий вузол з частотою 5+6=11

```
          (11)
         /    \
      A:5     (6)
             /   \
          R:2    (4)
                /   \
             (2)    B:2
            /   \
         C:1    D:1
```

Генерація кодів із дерева

```
Алгоритм GenerateHuffmanCodes(root):
    Вхід:
        root - корінь дерева Хаффмана
    Вихід:
        codeTable - таблиця кодів для кожного символу

    codeTable ← новий словник

    // Допоміжна рекурсивна функція
    процедура GenerateCodesRecursive(node, currentCode):
        якщо node = null:
            повернути

        // Якщо це листок - зберігаємо код символу
        якщо node.isLeaf():
            якщо currentCode = "":
                // Спеціальний випадок: єдиний символ
                codeTable[node.symbol] ← "0"
            інакше:
                codeTable[node.symbol] ← currentCode
            повернути

        // Рекурсивно обходимо ліве піддерево (додаємо '0')
        GenerateCodesRecursive(node.leftChild, currentCode + "0")
```

```
        // Рекурсивно обходимо праве піддерево (додаємо '1')
        GenerateCodesRecursive(node.rightChild, currentCode + "1")

    // Запускаємо рекурсію з кореня
    GenerateCodesRecursive(root, "")

    повернути codeTable
```

Для нашого прикладу отримаємо коди:

- A: 0

- R: 10

- C: 1100

- D: 1101

- B: 111

Процес кодування

```
Алгоритм EncodeData(data, codeTable):
    Вхід:
        data - масив символів для кодування
        codeTable - таблиця кодів Хаффмана
    Вихід:
        encodedBits - закодована бітова послідовність

    encodedBits ← порожній рядок

    для кожного символу symbol в data:
        якщо symbol існує в codeTable:
            encodedBits ← encodedBits + codeTable[symbol]
        інакше:
            помилка "Символ не знайдено в таблиці кодів"

    повернути encodedBits
```

Процес декодування

```
Алгоритм DecodeData(encodedBits, root):
    Вхід:
        encodedBits - закодована бітова послідовність
        root - корінь дерева Хаффмана
    Вихід:
        decodedData - декодовані дані

    decodedData ← порожній масив
    currentNode ← root
```

```
    для кожного біта bit в encodedBits:
        // Рухаємось вниз по дереву
        якщо bit = '0':
            currentNode ← currentNode.leftChild
        інакше:
            currentNode ← currentNode.rightChild

        // Якщо досягли листка - знайшли символ
        якщо currentNode.isLeaf():
            додати currentNode.symbol до decodedData
            currentNode ← root  // Повертаємось до кореня

    повернути decodedData
```

Оптимальність алгоритму

Алгоритм Хаффмана гарантує побудову оптимального префіксного коду. Це означає, що серед усіх можливих префіксних кодів, код Хаффмана дає найменшу середню довжину закодованого повідомлення.

Доведення оптимальності базується на двох ключових властивостях: 1. Символи з найменшими частотами повинні мати найдовшу довжину коду 2. Два символи з найменшими частотами можуть відрізнятися лише останнім бітом

Варіації та розширення

1. Адаптивний алгоритм Хаффмана

```
Алгоритм AdaptiveHuffmanEncode(data):
    Вхід:
        data - потік даних
    Вихід:
        encodedStream - закодований потік

    // Ініціалізація з порожнім деревом
    tree ← InitializeEmptyTree()
    encodedStream ← порожній потік

    для кожного символу symbol в data:
        якщо symbol вже в дереві:
            // Кодуємо існуючий символ
            code ← GetCode(tree, symbol)
            додати code до encodedStream
        інакше:
            // Кодуємо спеціальний NYT (Not Yet Transmitted) код
```

```
            nytCode ← GetNYTCode(tree)
            додати nytCode до encodedStream
            додати symbol як сирі біти до encodedStream

            // Додаємо новий символ до дерева
            AddSymbolToTree(tree, symbol)

        // Оновлюємо дерево після кожного символу
        UpdateTree(tree, symbol)

    повернути encodedStream
```

Область застосування

Алгоритм Хаффмана широко використовується в:

- Архіватори та компресори - ZIP, GZIP, BZIP2 використовують кодування Хаффмана

- Формати зображень - JPEG використовує модифікований Хаффман для стиснення коефіцієнтів

- Мультимедіа кодеки - MP3, AAC для аудіо, H.264 для відео

- Факсимільна передача - стандарти факсів використовують коди Хаффмана

- Мережеві протоколи - HTTP/2 HPACK для стиснення заголовків

- Файлові системи - деякі ФС використовують Хаффмана для метаданих

- Передача даних - модеми та інші пристрої зв'язку

- Бази даних - стиснення індексів та даних

Приклади застосування

Приклад 1: Стиснення текстового файлу

Для англійського тексту типові частоти: - 'e': 12.7% → код '10' - 't': 9.1% → код '110' - 'a': 8.2% → код '010' - 'z': 0.07% → код '111110101'

Приклад 2: Стиснення чорно-білого зображення

Для зображення з переважно білим фоном: - Білий піксель: 85% → код '0' - Чорний піксель: 15% → код '1'

Приклад 3: Стиснення JSON

Часті токени JSON отримують короткі коди: - '":': код '10' - '","': код '110' - '{}': код '1110'

Переваги та недоліки

Переваги

- Оптимальність - гарантує найкоротший середній код серед префіксних кодів

- Простота декодування - не потребує роздільників між символами

- Відсутність втрат - точне відновлення початкових даних

- Ефективність - добре працює для даних з нерівномірним розподілом

- Теоретична обґрунтованість - математично доведена оптимальність

- Широка підтримка - стандарт де-факто для багатьох застосувань

Недоліки

- Необхідність передачі дерева - збільшує розмір для малих файлів

- Два проходи по даних - для статичного Хаффмана

- Цілочисельні довжини кодів - не може використовувати дробові біти

- Чутливість до помилок - один пошкоджений біт може зруйнувати декодування

- Неефективність для рівномірних розподілів - коли всі символи мають близькі частоти

- Відсутність контексту - не враховує залежності між символами

Контрольні запитання

1. Що таке префіксний код і чому він важливий для алгоритму Хаффмана?

2. Поясніть принцип жадібності в алгоритмі Хаффмана

3. Чому використовується пріоритетна черга при побудові дерева?

4. Як забезпечується оптимальність коду Хаффмана?

5. Що відбувається, якщо всі символи мають однакову частоту?

6. Як працює декодування без знання меж між кодами?

7. Чим відрізняється статичний та адаптивний Хаффман?

8. Як обробляється випадок з одним унікальним символом?

9. Яка часова складність побудови дерева Хаффмана?

10. Як мінімізувати накладні витрати на зберігання дерева?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму кодування Хаффмана для стиснення даних

2. Мета

Освоїти принципи префіксного кодування, навчитися будувати оптимальні коди, реалізувати повний цикл стиснення та декомпресії даних.

3. Завдання

- Реалізувати структуру вузла дерева Хаффмана
- Створити функцію аналізу частот символів
- Реалізувати побудову дерева з використанням пріоритетної черги
- Створити функцію генерації кодів обходом дерева
- Реалізувати процедури кодування та декодування
- Додати серіалізацію/десеріалізацію дерева
- Створити візуалізацію побудованого дерева
- Провести аналіз ефективності для різних типів даних

4. Опис

Лабораторна робота передбачає повну реалізацію алгоритму Хаффмана від аналізу вхідних даних до створення стиснутого файлу з можливістю його розпакування. Особлива увага приділяється коректній роботі з бітами та серіалізації структури дерева.

5. Порядок виконання

- Створити клас HuffmanNode для представлення вузлів дерева
- Реалізувати клас PriorityQueue або використати готову реалізацію
- Написати функцію buildFrequencyTable для аналізу частот
- Імплементувати функцію buildHuffmanTree
- Створити функцію generateCodes для рекурсивного обходу
- Реалізувати класи BitWriter та BitReader для роботи з бітами
- Написати функції encode та decode
- Додати методи serializeTree та deserializeTree
- Створити візуалізатор дерева (консольний або графічний)
- Реалізувати повний компресор з інтерфейсом командного рядка

6. Результат

Готова програма повинна: - Приймати файл на вхід та створювати стиснуту версію - Зберігати структуру дерева разом зі стисненими даними - Коректно

розпаковувати стиснені файли - Показувати статистику стиснення (розмір до/після, коефіцієнт) - Візуалізувати побудоване дерево Хаффмана - Працювати з різними типами файлів (текст, бінарні дані)

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас HuffmanNode з конструкторами та методами

- Модуль побудови дерева з детальними коментарями

- Функції кодування та декодування

- Модуль серіалізації дерева

- Клас для роботи з бітовими потоками

- Головна програма з інтерфейсом

2. Продемонструвати роботу додатку:

- Стиснення текстового файлу з показом коефіцієнта

- Візуалізація дерева для невеликого прикладу

- Декомпресія та перевірка ідентичності

- Робота з різними типами файлів

- Аналіз залежності ефективності від типу даних

- Обробка граничних випадків

3. Бути готовим відповісти на питання:

- Як вибирається порядок об’єднання вузлів?

- Чому код є префіксним?

- Як зберегти дерево мінімальним обсягом?

- Що робити з файлами, де всі байти різні?

- Як оцінити теоретичну межу стиснення?

Тести

1. Яка структура даних використовується для ефективної побудови дерева Хаффмана?

1) Стек

2) Черга FIFO

3) Пріоритетна черга (мін-купа)

4) Хеш-таблиця

2. Скільки вузлів буде в дереві Хаффмана для алфавіту з n символів?

1) n

2) 2n

3) 2n - 1

4) n^2

3. Що визначає довжину коду символу в дереві Хаффмана?

1) Частота символу

2) Глибина символу в дереві

3) Порядок символу в алфавіті

4) Розмір файлу

4. Який символ матиме найкоротший код?

1) Перший в алфавіті

2) Останній в алфавіті

3) З найбільшою частотою

4) З найменшою частотою

5. Коли алгоритм Хаффмана дає найгірший результат?

1) Великі файли

2) Малі файли

3) Рівномірний розподіл частот

4) Дуже нерівномірний розподіл

Відповіді:

1. c) - Пріоритетна черга дозволяє ефективно вибирати вузли з мінімальною частотою

2. c) - Дерево має n листків та n-1 внутрішніх вузлів

3. b) - Довжина коду = кількість ребер від кореня до листка

4. c) - Найчастіший символ розміщується ближче до кореня

5. c) - При однакових частотах немає виграшу від змінної довжини

Тести підвищеної складності

1. Яка мінімальна середня довжина біт на символ для алфавіту з 8 рівноймовірних символів?

1) 2

2) 3

3) 4

4) 8

2. При об’єднанні двох вузлів з частотами f1 та f2, нова частота буде:

1) max(f1, f2)

2) min(f1, f2)

3) f1 + f2

4) f1 × f2

3. Скільки бітів потрібно для кодування файлу з ентропією H біт/символ та n символів?

1) Рівно n × H

2) Між n × H та n × (H + 1)

3) Менше n × H

4) Більше n × (H + 2)

4. Яка максимальна глибина дерева Хаффмана для n символів?

1) $\log_2 n$

2) n/2

3) n - 1

4) 2^n

5. Коли два символи можуть мати коди однакової довжини?

1) Ніколи

2) Коли мають однакові частоти

3) Завжди

4) Коли їх частоти відрізняються на 1

6. Скільки різних дерев Хаффмана можна побудувати для заданого набору частот?

1) Рівно одне

2) Не більше n

3) Може бути багато

4) 2^n

7. Яка властивість гарантує однозначність декодування?

1) Сортування за частотою

2) Префіксність коду

3) Бінарність дерева

4) Використання 0 та 1

8. Для передачі структури дерева з n листками потрібно мінімум:

1) n біт

2) n log n біт

3) 2n - 1 біт

4) n^2 біт

9. При додаванні нового символу з нульовою частотою до готового дерева:

1) Дерево не зміниться

2) Потрібно повністю перебудувати

3) Додати як брата найрідкіснішого

4) Неможливо додати

10. Адаптивний Хаффман порівняно зі статичним:

1) Завжди ефективніший

2) Потребує лише один прохід

3) Простіший в реалізації

4) Дає гірше стиснення

Відповіді:

1. b) 3 - $\log_2(8) = 3$ біти для рівноймовірних символів

2. c) f1 + f2 - частоти додаються при об’єднанні

3. b) - За теоремою Шеннона середня довжина між H та H+1

4. c) n - 1 - у виродженому випадку лінійного дерева

5. c) - Різні символи можуть мати однакову довжину коду

6. c) - При однакових частотах можливі різні порядки об’єднання

7. b) - Префіксна властивість забезпечує однозначність

8. c) 2n - 1 - по біту на кожен вузол дерева

9. b) - Додавання нового символу змінює оптимальну структуру

10. b) - Не потребує попереднього підрахунку частот

Завдання для самостійної роботи

1. Реалізуйте канонічний код Хаффмана, який дозволяє зберігати лише довжини кодів замість всього дерева.

2. Створіть версію алгоритму для алфавітів розміром більше 256 символів (Unicode).

3. Розробіть гібридний алгоритм, що поєднує Хаффмана з RLE (Run-Length Encoding).

4. Імплементуйте обмеження максимальної довжини коду з мінімальною втратою оптимальності.

5. Створіть паралельну версію кодування для багатоядерних процесорів.

6. Реалізуйте стиснення Хаффмана для потокових даних з обмеженим буфером.

7. Розробіть алгоритм вибору оптимального розміру блоку для блочного Хаффмана.

8. Створіть адаптивний Хаффман з періодичним перебудовуванням дерева.

9. Імплементуйте двопрохідний Хаффман з попереднім аналізом для вибору стратегії.

10. Розробіть систему автоматичного визначення типу даних для оптимального налаштування.

Теми лабораторних робіт

1. Розробка універсального архіватора з підтримкою різних режимів Хаффмана.

2. Створення плагіна стиснення для текстового редактора з миттєвою компресією.

3. Імплементація мережевого протоколу з вбудованим стисненням Хаффмана.

4. Розробка системи стиснення логів з оптимізацією для повторюваних патернів.

5. Створення бібліотеки швидкого Хаффман-кодування з SIMD-оптимізаціями.

6. Реалізація файлової системи зі прозорим стисненням на основі Хаффмана.

7. Розробка інструменту аналізу ефективності стиснення для різних типів даних.

8. Створення навчального симулятора для візуалізації роботи алгоритму Хаффмана.

9. Імплементація стиснення баз даних з селективним застосуванням Хаффмана.

10. Розробка криптографічно захищеного Хаффмана з шифруванням структури дерева.

10.4.4. Алгоритм Фано

Теоретичні відомості

Алгоритм Фано — це метод побудови префіксних кодів змінної довжини, розроблений Робертом Фано в 1949 році. Основна ідея алгоритму полягає в створенні двійкового коду для символів на основі їх частот або ймовірностей появи в повідомленні. Алгоритм належить до класу методів ентропійного кодування, які намагаються наблизити середню довжину коду до ентропії джерела інформації.

Нагадаємо, що префіксний код — це такий код, в якому жодне кодове слово не є префіксом (початком) іншого кодового слова.

Наприклад, якщо символ A кодується як “01”, то жоден інший символ не може мати код, що починається з “01”, такий як “010” або “011”. Ця властивість забезпечує однозначність декодування без необхідності використання роздільників між кодами.

Основний принцип алгоритму Фано базується на ідеї рекурсивного поділу множини символів на дві підмножини з приблизно рівними сумарними ймовірностями. Символи з більшими частотами отримують коротші коди, а символи з меншими частотами — довші коди, що забезпечує ефективне стиснення даних.

Побудова дерева кодування за алгоритмом Фано складається з таких етапів:

Етап 1: Підготовка даних

Спочатку необхідно визначити частоту появи кожного символу у вхідному повідомленні. Частота — це кількість разів, яку символ зустрічається в тексті. Після підрахунку частот, символи впорядковуються за спаданням їх частот. Це впорядкування є критично важливим для ефективної роботи алгоритму.

Етап 2: Рекурсивний поділ

Основна операція алгоритму — поділ впорядкованої множини символів на дві групи. Мета поділу — отримати дві групи з якомога ближчими сумарними частотами. Для цього:

- Обчислюється загальна сума частот усіх символів у поточній групі

- Знаходиться точка поділу, де сума частот лівої частини найближча до половини загальної суми

- Символи діляться на дві групи відповідно до знайденої точки

Етап 3: Присвоєння бітів

Після поділу групи на дві підгрупи: - Усім символам лівої підгрупи додається біт “0” до їх коду - Усім символам правої підгрупи додається біт “1” до їх коду

Етап 4: Рекурсивне застосування

Процес поділу та присвоєння бітів рекурсивно застосовується до кожної підгрупи, доки кожна підгрупа не міститиме лише один символ. Коли підгрупа містить лише один символ, формування коду для цього символу завершено.

Розглянемо детальний приклад побудови дерева кодування для повідомлення “ABRACADABRA”.

Крок 1: Підрахунок частот

Спочатку підраховуємо, скільки разів кожен символ зустрічається в повідомленні: - A: 5 разів - B: 2 рази - R: 2 рази - C: 1 раз - D: 1 раз

Крок 2: Впорядкування за спаданням частот

Впорядковуємо символи: A(5), B(2), R(2), C(1), D(1)

Крок 3: Побудова дерева

Початкова група: {A(5), B(2), R(2), C(1), D(1)} Загальна сума частот: 5 + 2 + 2 + 1 + 1 = 11

Перший поділ:

- Половина суми: 11 / 2 = 5.5
- Перевіряємо можливі точки поділу:
    - Після A: ліва сума = 5, права сума = 6
    - Після B: ліва сума = 7, права сума = 4
    - Після R: ліва сума = 9, права сума = 2

Найкращий поділ — після A, оскільки |5 - 5.5| = 0.5 є мінімальним.

Результат першого поділу:

- Ліва група: {A(5)} → отримує біт “0”
- Права група: {B(2), R(2), C(1), D(1)} → отримує біт “1”

Ліва група містить лише один символ, тому код для A завершено: A = “0”

Обробка правої групи {B(2), R(2), C(1), D(1)}:

- Сума частот: 2 + 2 + 1 + 1 = 6
- Половина: 6 / 2 = 3
- Можливі точки поділу:
    - Після B: ліва сума = 2, права сума = 4
    - Після R: ліва сума = 4, права сума = 2

Різниця |2 - 3| = 1 та |4 - 3| = 1 однакові. У таких випадках обираємо поділ, що створює більш збалансовані групи за кількістю елементів. Обираємо поділ після R.

Результат:

- Ліва підгрупа: {B(2), R(2)} → додається біт “0”
- Права підгрупа: {C(1), D(1)} → додається біт “1”

Обробка підгрупи {B(2), R(2)}:

- Сума: 4, половина: 2
- Поділ після B: ліва = 2, права = 2 (ідеальний поділ)
- B отримує “0”, R отримує “1”

Обробка підгрупи {C(1), D(1)}:

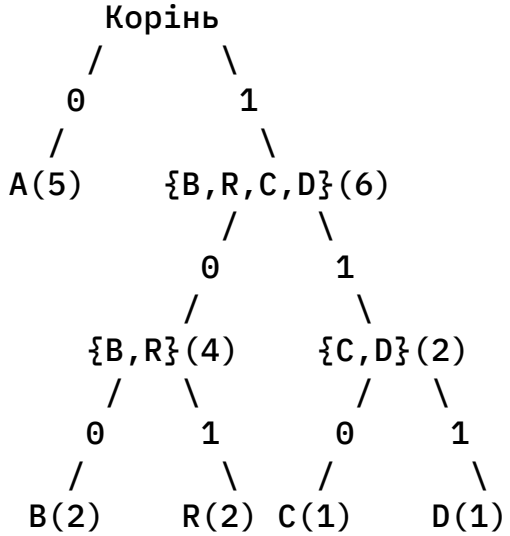
- Сума: 2, половина: 1

- Поділ після C: ліва = 1, права = 1 (ідеальний поділ)

- C отримує “0”, D отримує “1”

Фінальні коди: - A: “0” - B: “100” - R: “101” - C: “110” - D: “111”

Візуалізація дерева кодування



Алгоритм побудови коду Фано

```
Алгоритм FanoCoding(symbols, frequencies, codes, left, right, prefix)
:
    Вхід:
        symbols - масив символів
        frequencies - масив частот символів
        codes - масив для збереження кодів
        left, right - межі поточної групи
        prefix - поточний префікс коду

    // Базовий випадок: один символ у групі
    якщо left == right:
        codes[left] = prefix
        повернути

    // Обчислення загальної суми частот у групі
    total_sum = 0
    для i від left до right:
        total_sum = total_sum + frequencies[i]

    // Знаходження оптимальної точки поділу
    half_sum = total_sum / 2
    current_sum = 0
    split_point = left
```

```
    min_difference = нескінченність

    для i від left до right - 1:
        current_sum = current_sum + frequencies[i]
        difference = |current_sum - half_sum|

        якщо difference < min_difference:
            min_difference = difference
            split_point = i

    // Рекурсивні виклики для лівої та правої груп
    FanoCoding(symbols, frequencies, codes, left, split_point, prefix
+ "0")
    FanoCoding(symbols, frequencies, codes, split_point + 1, right, p
refix + "1")
```

Розглянемо роботу алгоритму на прикладі кодування слова “MISSISSIPPI”.

Крок 1: Аналіз вхідних даних

Підрахунок частот:

- M: 1

- I: 4

- S: 4

- P: 2

Крок 2: Сортування

Впорядкування за спаданням: I(4), S(4), P(2), M(1)

Крок 3: Покрокове виконання алгоритму

Виклик 1: FanoCoding(вся група, prefix = ““)

- Група: {I(4), S(4), P(2), M(1)}

- Загальна сума: 11 - Половина: 5.5

- Перевірка точок поділу:

    - Після I: ліва = 4, права = 7, різниця = |4 - 5.5| = 1.5

    - Після S: ліва = 8, права = 3, різниця = |8 - 5.5| = 2.5

    - Після P: ліва = 10, права = 1, різниця = |10 - 5.5| = 4.5

- Обрано поділ після I

Виклик 2: FanoCoding({I}, prefix = “0”)

- База рекурсії: I = “0”

Виклик 3: FanoCoding({S, P, M}, prefix = “1”)

- Група: {S(4), P(2), M(1)}

- Сума: 7, половина: 3.5

- Точки поділу:

    - Після S: ліва = 4, права = 3, різниця = |4 - 3.5| = 0.5

    - Після P: ліва = 6, права = 1, різниця = |6 - 3.5| = 2.5

- Обрано поділ після S

Виклик 4: FanoCoding({S}, prefix = “10”)

- База рекурсії: S = “10”

Виклик 5: FanoCoding({P, M}, prefix = “11”)

- Група: {P(2), M(1)} - Сума: 3, половина: 1.5

- Після P: ліва = 2, права = 1, різниця = |2 - 1.5| = 0.5

- Поділ після P

Виклик 6: FanoCoding({P}, prefix = “110”)

- База рекурсії: P = “110”

Виклик 7: FanoCoding({M}, prefix = “111”)

- База рекурсії: M = “111”

Результуючі коди:

- I: “0”

- S: “10”

- P: “110”

- M: “111”

Кодування слова “MISSISSIPPI”: M(111) I(0) S(10) S(10) I(0) S(10) S(10) I(0) P(110) P(110) I(0)

Закодоване повідомлення: 11101010010100110110

Алгоритм декодування

```
Алгоритм FanoDecode(encoded_message, code_table):
    Вхід:
        encoded_message - закодоване повідомлення (рядок бітів)
        code_table - таблиця відповідності кодів символам
    Вихід:
        decoded_message - декодоване повідомлення
```

```
    decoded_message = ""
    current_code = ""

    для кожен bit в encoded_message:
        current_code = current_code + bit

        // Перевірка чи current_code є в таблиці
        для кожен (code, symbol) в code_table:
            якщо current_code == code:
                decoded_message = decoded_message + symbol
                current_code = ""
                перервати

    повернути decoded_message
```

Повний алгоритм з підготовкою даних

```
Алгоритм CompleteFanoAlgorithm(message):
    Вхід: message - повідомлення для кодування
    Вихід: encoded_message - закодоване повідомлення
           code_table - таблиця кодів

    // Етап 1: Підрахунок частот
    frequency_map = новий словник

    для кожен symbol в message:
        якщо symbol є в frequency_map:
            frequency_map[symbol] = frequency_map[symbol] + 1
        інакше:
            frequency_map[symbol] = 1

    // Етап 2: Створення масивів символів та частот
    n = кількість унікальних символів
    symbols = новий масив[n]
    frequencies = новий масив[n]

    i = 0
    для кожен (symbol, frequency) в frequency_map:
        symbols[i] = symbol
        frequencies[i] = frequency
        i = i + 1

    // Етап 3: Сортування за спаданням частот
    для i від 0 до n-2:
        для j від 0 до n-2-i:
```

```
            якщо frequencies[j] < frequencies[j+1]:
                обміняти(frequencies[j], frequencies[j+1])
                обміняти(symbols[j], symbols[j+1])

    // Етап 4: Побудова кодів
    codes = новий масив[n] рядків
    FanoCoding(symbols, frequencies, codes, 0, n-1, "")

    // Етап 5: Створення таблиці кодів
    code_table = новий словник
    для і від 0 до n-1:
        code_table[symbols[i]] = codes[i]

    // Етап 6: Кодування повідомлення
    encoded_message = ""
    для кожен symbol в message:
        encoded_message = encoded_message + code_table[symbol]

    повернути encoded_message, code_table
```

Область застосування

Алгоритм Фано знаходить застосування в різних сферах обробки та передачі інформації:

- Стиснення даних: Алгоритм використовується для зменшення обсягу даних шляхом присвоєння коротших кодів частішим символам. Це особливо ефективно для текстових файлів з нерівномірним розподілом символів.

- Телекомунікації: У системах передачі даних алгоритм дозволяє зменшити кількість бітів, необхідних для передачі повідомлення, що підвищує ефективність використання каналу зв'язку.

- Архівація файлів: Хоча алгоритм Фано не є найефективнішим серед методів стиснення, він може використовуватися як компонент більш складних систем архівації або для попередньої обробки даних.

- Навчальні цілі: Завдяки своїй простоті та наочності, алгоритм широко використовується для навчання основам теорії інформації та методам стиснення даних.

- Вбудовані системи: У системах з обмеженими обчислювальними ресурсами алгоритм Фано може бути кращим вибором через свою простоту реалізації порівняно з більш складними алгоритмами.

Приклади застосування

Приклад 1: Стиснення текстового файлу

Розглянемо текстовий файл з англійським текстом, де літера ‘e’ зустрічається найчастіше (приблизно 12%), ‘t’ — 9%, ‘a’ — 8%, і так далі. Застосування алгоритму Фано дозволить присвоїти:

- Літері ‘e’ — код довжиною 3 біти

- Літері ‘t’ — код довжиною 3-4 біти

- Рідкісним літерам як ‘z’ або ‘q’ — коди довжиною 7-8 бітів

Це забезпечить середню довжину коду близько 4.5 бітів на символ замість 8 бітів при використанні ASCII.

Приклад 2: Кодування команд у системі управління

У системі дистанційного управління, де певні команди використовуються частіше:

- “Вперед” (40% використання) — код “0”

- “Назад” (25%) — код “10”

- “Ліворуч” (20%) — код “110”

- “Праворуч” (15%) — код “111”

Це дозволяє передавати найчастіші команди швидше, економлячи енергію та час передачі.

Приклад 3: Стиснення логів веб-сервера

У файлах логів веб-сервера часто повторюються певні IP-адреси, коди відповідей (200, 404), методи запитів (GET, POST). Алгоритм Фано може ефективно стискати такі дані, присвоюючи короткі коди частим елементам:

- Код відповіді “200” — “0”

- Метод “GET” — “10”

- Код “404” — “110”

- Метод “POST” — “111”

Переваги та недоліки

Переваги:

- Простота реалізації: Алгоритм має інтуїтивно зрозумілу логіку та легко реалізується з використанням рекурсії

- Префіксність коду: Гарантує однозначність декодування без необхідності роздільників

- Швидкість роботи: Час побудови коду O(n log n), що є ефективним для більшості практичних застосувань

- Адаптивність: Легко модифікується для роботи з динамічними частотами символів

- Низькі вимоги до пам’яті: Не потребує складних структур даних для зберігання

Недоліки:

- Субоптимальність: Не завжди створює код з мінімальною середньою довжиною (на відміну від алгоритму Хаффмана)

- Чутливість до порядку: При однакових частотах результат залежить від початкового порядку символів

- Необхідність передачі таблиці кодів: Декодер повинен знати таблицю відповідності, що збільшує накладні витрати

- Неефективність для рівномірних розподілів: При однакових частотах всіх символів алгоритм не дає виграшу в стисненні

- Відсутність оптимізації для контексту: Не враховує залежності між сусідніми символами

Контрольні запитання

1. Що таке префіксний код і яка його основна властивість?

2. Який критерій використовується для поділу групи символів на дві підгрупи в алгоритмі Фано?

3. Чому символи необхідно впорядковувати за спаданням частот перед побудовою коду?

4. Яка часова складність алгоритму Фано?

5. У чому полягає основна відмінність між алгоритмами Фано та Хаффмана?

6. Як відбувається процес декодування повідомлення, закодованого алгоритмом Фано?

7. Які фактори впливають на ефективність стиснення алгоритмом Фано?

8. Чи можуть два різні символи мати коди однакової довжини в алгоритмі Фано?

9. Як алгоритм обробляє символи з однаковими частотами?

10. Які модифікації алгоритму Фано можуть покращити його ефективність?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Фано для стиснення текстових даних

2. Мета

Освоїти принципи побудови префіксних кодів, навчитися реалізовувати алгоритм Фано та аналізувати ефективність стиснення даних

3. Завдання

- Реалізувати функцію підрахунку частот символів

- Реалізувати сортування символів за частотами

- Реалізувати рекурсивну функцію побудови кодів Фано

- Створити функції кодування та декодування повідомлень

- Реалізувати візуалізацію дерева кодування

- Провести аналіз ефективності стиснення для різних типів текстів

4. Опис

Лабораторна робота передбачає розробку програми, яка реалізує повний цикл роботи з алгоритмом Фано: від аналізу вхідного тексту до побудови кодів, кодування, декодування та аналізу ефективності. Програма повинна працювати з текстовими файлами різних розмірів та мати зручний інтерфейс для демонстрації результатів.

5. Порядок виконання

- Створити структури даних для зберігання символів, частот та кодів

- Реалізувати функцію аналізу частот символів у тексті

- Реалізувати алгоритм сортування (рекомендується quicksort або mergesort)

- Написати рекурсивну функцію побудови кодів Фано

- Створити функцію кодування тексту з використанням побудованої таблиці

- Реалізувати функцію декодування закодованого повідомлення

- Додати функції для обчислення коефіцієнта стиснення та середньої довжини коду

- Реалізувати візуалізацію дерева кодування (текстову або графічну)

- Створити тестові приклади для перевірки коректності роботи

- Провести експерименти з різними типами текстів

6. Результат

Готова програма з документацією, яка демонструє роботу алгоритму Фано, включаючи візуалізацію процесу побудови коду та статистику ефективності стиснення

Після виконання лабораторної роботи студент повинен надати:

- **Вихідний код програми, де чітко виділені:**
- Модуль аналізу частот символів
- Функція сортування за частотами
- Рекурсивна функція побудови кодів Фано
- Функція кодування повідомлення
- Функція декодування повідомлення
- Модуль візуалізації дерева
- Модуль обчислення статистики
- **Продемонструвати роботу додатку:**
- Завантаження та аналіз текстового файлу
- Виведення таблиці частот символів
- Покрокова демонстрація побудови дерева кодування
- Виведення таблиці символів та їх кодів
- Кодування та декодування тестового повідомлення
- Порівняння розмірів вихідного та закодованого файлів
- Обчислення та виведення коефіцієнта стиснення
- **Бути готовим відповісти на питання:**
- Як визначається точка поділу при побудові коду?
- Чому результат алгоритму може бути субоптимальним?
- Як працює декодування префіксного коду?
- Які модифікації можна внести для покращення алгоритму?
- Як алгоритм поводиться з символами однакової частоти?

Тести

1. Основна властивість префіксного коду полягає в тому, що:

1) Всі коди мають однакову довжину

2) Жоден код не є початком іншого коду

3) Коди складаються лише з одиниць

4) Довжина коду пропорційна частоті символу

2. При побудові коду Фано символи спочатку впорядковуються за:

1) Алфавітним порядком

2) Зростанням частот

3) Спаданням частот

4) Випадковим порядком

3. Критерієм поділу групи символів на дві підгрупи є:

1) Рівна кількість символів у кожній підгрупі

2) Мінімальна різниця між сумами частот підгруп

3) Максимальна різниця між сумами частот підгруп

4) Випадковий поділ

4. Часова складність алгоритму Фано становить:

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) O(2^n)

5. Алгоритм Фано гарантує:

1) Оптимальний код з мінімальною середньою довжиною

2) Префіксний код

3) Код фіксованої довжини

4) Мінімальний час декодування

Відповіді: 1. b) 2. c) 3. b) 4. b) 5. b)

Тести підвищеної *складності*

1. Для символів з частотами A(50), B(25), C(15), D(10) код символу A матиме довжину:

1) 1 біт

2) 2 біти

3) 3 біти

4) 4 біти

2. При кодуванні 8 символів з однаковими частотами алгоритм Фано створить коди довжиною:

1) Рівно 3 біти для всіх символів

2) Від 2 до 4 бітів

3) Рівно 4 біти для всіх символів

4) Від 3 до 5 бітів

3. Якщо два символи мають однакові частоти, то їх коди:

1) Обов’язково матимуть однакову довжину

2) Можуть мати різну довжину

3) Завжди відрізнятимуться лише останнім бітом

4) Будуть ідентичними

4. Коефіцієнт стиснення 0.75 означає, що:

1) Файл стиснуто на 75%

2) Стиснутий файл займає 75% від оригіналу

3) Економія складає 75%

4) Файл збільшився на 75%

5. При декодуванні послідовності “1001” з кодами A=“0”, B=“10”, C=“11” результат буде:

1) BAA

2) BC

3) CA

4) Неможливо декодувати

6. Середня довжина коду L обчислюється за формулою:

1) $L = \Sigma(p_i \times l_i)$, де p_i - ймовірність, l_i - довжина коду

2) $L = \Sigma(l_i) / n$

3) $L = \max(l_i)$

4) $L = \min(l_i)$

7. Ентропія джерела з двома рівноймовірними символами дорівнює:

1) 0 біт

2) 0.5 біт

3) 1 біт

4) 2 біти

8. Алгоритм Фано створює субоптимальний код тому, що:

1) Не враховує частоти символів

2) Використовує жадібний підхід при поділі

3) Не створює префіксні коди

4) Працює занадто швидко

9. Для ефективного стиснення алгоритмом Фано необхідно:

1) Рівномірний розподіл частот

2) Нерівномірний розподіл частот

3) Малий розмір алфавіту

4) Великий розмір повідомлення

10. При передачі закодованого повідомлення необхідно також передати:

1) Лише закодовані дані

2) Таблицю відповідності символів та кодів

3) Алгоритм декодування

4) Частоти всіх символів алфавіту

Відповіді:

1. a) При частоті 50% символ A буде відділений на першому кроці і отримає код довжиною 1 біт

2. a) При однакових частотах алгоритм створює збалансоване дерево глибиною $\log_2(8) = 3$

3. b) Символи з однаковими частотами можуть опинитися в різних групах при поділі

4. b) Коефіцієнт стиснення показує відношення розміру стиснутого файлу до оригіналу

5. a) Декодування: 10(B) 0(A) 1 → знову перевіряємо → 10(B) 0(A) 1 не є повним кодом, але якщо продовжити читати, то отримаємо 10(B) 0(A) 0(A) при наступному біті 0

6. a) Середня довжина коду - це математичне сподівання довжини коду

7. c) $H = -0.5 \times \log_2(0.5) - 0.5 \times \log_2(0.5) = 0.5 + 0.5 = 1$ біт

8. b) Жадібний підхід не гарантує глобального оптимуму

9. b) Нерівномірний розподіл дозволяє присвоювати короткі коди частим символам

10. b) Без таблиці кодів декодер не зможе відновити вихідне повідомлення

Завдання для самостійної роботи

1. Реалізуйте модифікацію алгоритму Фано, яка враховує не лише частоти окремих символів, але й частоти пар символів (біграм). Порівняйте ефективність стиснення з базовим алгоритмом.

2. Створіть програму для порівняння ефективності алгоритмів Фано та Хаффмана на різних типах даних (текст, числові дані, випадкові дані). Побудуйте графіки залежності коефіцієнта стиснення від ентропії джерела.

3. Розробіть адаптивну версію алгоритму Фано, яка перебудовує таблицю кодів під час обробки потоку даних. Дослідіть вплив частоти оновлення на ефективність стиснення.

4. Реалізуйте паралельну версію алгоритму Фано з використанням багатопоточності. Розділіть процес побудови дерева на незалежні підзадачі та виміряйте прискорення.

5. Створіть візуалізатор, який покроково демонструє процес побудови дерева кодування Фано з анімацією поділу груп та присвоєння кодів.

6. Дослідіть вплив різних стратегій вирішення конфліктів при однакових сумах частот на якість результуючого коду. Реалізуйте кілька стратегій та порівняйте їх.

7. Розробіть систему стиснення зображень, використовуючи алгоритм Фано для кодування значень пікселів або їх різниць. Порівняйте з існуючими методами.

8. Створіть гібридний алгоритм, який автоматично вибирає між алгоритмами Фано, Хаффмана та арифметичним кодуванням залежно від характеристик вхідних даних.

9. Реалізуйте механізм виявлення та виправлення помилок для повідомлень, закодованих алгоритмом Фано, використовуючи додаткові контрольні біти.

10. Дослідіть можливість використання алгоритму Фано для стиснення потокового аудіо. Реалізуйте прототип з буферизацією та аналізом ефективності в реальному часі.

Теми лабораторних робіт

1. Розробка універсального компресора файлів на основі алгоритму Фано з підтримкою різних типів даних та графічним інтерфейсом користувача.

2. Створення системи оптимізації передачі даних у мережі з використанням адаптивного алгоритму Фано для динамічного стиснення трафіку.

3. Реалізація криптографічно стійкої системи стиснення, яка комбінує алгоритм Фано з методами шифрування для забезпечення конфіденційності.

4. Розробка інтелектуальної системи вибору оптимального алгоритму стиснення (Фано, Хаффман, LZW) на основі аналізу характеристик даних.

5. Створення розподіленої системи стиснення великих обсягів даних з використанням алгоритму Фано та технологій MapReduce або Spark.

6. Реалізація системи стиснення баз даних з використанням алгоритму Фано для оптимізації зберігання текстових полів та логів.

7. Розробка мобільного додатку для стиснення та передачі зображень з використанням модифікованого алгоритму Фано для різних колірних каналів.

8. Створення навчального середовища для вивчення алгоритмів стиснення з інтерактивною візуалізацією та можливістю покрокового виконання.

9. Реалізація системи резервного копіювання з інкрементальним стисненням на основі алгоритму Фано та дедуплікації даних.

10. Розробка плагіна для веб-браузера, який автоматично стискає та декодує текстовий контент веб-сторінок для економії трафіку.

10.4.4 Алгоритм DEFLATE

Теоретичні відомості

Алгоритм DEFLATE — це комбінований метод стиснення даних без втрат, розроблений Філом Кацом у 1991 році. Цей алгоритм поєднує дві потужні технології стиснення: алгоритм LZ77 (Лемпеля-Зіва 1977) для усунення надлишковості через пошук повторюваних послідовностей та алгоритм Хаффмана для оптимального кодування як літералів (окремих байтів), так і пар довжина-відстань, що представляють повтори.

Основна ідея DEFLATE полягає в двоетапному процесі стиснення. На першому етапі алгоритм LZ77 сканує вхідні дані, шукаючи послідовності байтів, які вже зустрічалися раніше. Коли знаходиться збіг, замість повторення всієї послідовності, алгоритм записує пару чисел: довжину збігу та відстань до попередньої появи цієї послідовності. На другому етапі результати роботи LZ77 (літерали та пари довжина-відстань) кодуються за допомогою алгоритму Хаффмана для досягнення максимального стиснення.

Унікальність DEFLATE полягає в тому, що він використовує два окремі дерева Хаффмана: одне для кодування літералів та довжин збігів, інше — для кодування відстаней. Це дозволяє оптимально адаптуватися до різних типів даних та досягати високих коефіцієнтів стиснення.

Структура алгоритму DEFLATE

Алгоритм DEFLATE працює з блоками даних. Кожен блок може бути одного з трьох типів:

- Нестиснений блок: дані копіюються без змін

- Блок з фіксованими кодами Хаффмана: використовуються попередньо визначені коди

- Блок з динамічними кодами Хаффмана: коди створюються спеціально для конкретних даних

Для кожного блоку з динамічними кодами будуються два дерева Хаффмана, які ми детально розглянемо.

Етап LZ77: Пошук збігів

Перш ніж перейти до побудови дерев Хаффмана, розглянемо роботу першого етапу — алгоритму LZ77.

```
Алгоритм LZ77_Compress(input_data, window_size, lookahead_size):
    Вхід:
        input_data - вхідні дані для стиснення
        window_size - розмір вікна пошуку (типово 32KB)
```

```
        lookahead_size - розмір буфера передогляду (типово 258 байт)
    Вихід:
        compressed - стиснені дані (літерали та пари довжина-відстань
)

    compressed = []
    position = 0

    поки position < довжина(input_data):
        // Визначаємо межі вікна пошуку
        window_start = максимум(0, position - window_size)
        window_end = position

        // Визначаємо межі буфера передогляду
        lookahead_start = position
        lookahead_end = мінімум(position + lookahead_size, довжина(in
put_data))

        // Шукаємо найдовший збіг
        best_length = 0
        best_distance = 0

        для i від window_start до window_end - 1:
            // Порівнюємо з поточною позицією
            match_length = 0

            поки (i + match_length < window_end та
                  lookahead_start + match_length < lookahead_end та
                  input_data[i + match_length] == input_data[lookahea
d_start + match_length]):
                match_length = match_length + 1

            // Оновлюємо найкращий збіг
            якщо match_length > best_length:
                best_length = match_length
                best_distance = position - i

        // Записуємо результат
        якщо best_length >= 3:  // Мінімальна довжина збігу
            compressed.додати(ПараДовжинаВідстань(best_length, best_d
istance))
            position = position + best_length
        інакше:
            compressed.додати(Літерал(input_data[position]))
            position = position + 1
```

```
    повернути compressed
```

Детальна побудова дерев Хаффмана в DEFLATE

У DEFLATE використовуються два основні дерева Хаффмана та одне допоміжне:

1. Дерево літералів/довжин: кодує значення від 0 до 285

- 0-255: літерали (байти)

- 256: маркер кінця блоку

- 257-285: коди довжин збігів (3-258 байт)

2. Дерево відстаней: кодує значення від 0 до 29

- Представляють відстані від 1 до 32768 байт

3. Дерево кодів довжин: використовується для компактного представлення перших двох дерев

Кодування довжин та відстаней

Перед побудовою дерев, довжини та відстані перетворюються в коди:

```
Структура LengthCode:
    base_length: базова довжина
    extra_bits: кількість додаткових бітів
    code: код Хаффмана

Алгоритм EncodeLengthDistance(raw_data):
    Вхід: raw_data - результат роботи LZ77
    Вихід: encoded_data - закодовані дані

    // Таблиця кодів довжин (257-285)
    length_codes = [
        {3, 0}, {4, 0}, {5, 0}, {6, 0}, {7, 0}, {8, 0}, {9, 0}, {10,
0},
        {11, 1}, {13, 1}, {15, 1}, {17, 1},
        {19, 2}, {23, 2}, {27, 2}, {31, 2},
        {35, 3}, {43, 3}, {51, 3}, {59, 3},
        {67, 4}, {83, 4}, {99, 4}, {115, 4},
        {131, 5}, {163, 5}, {195, 5}, {227, 5},
        {258, 0}
    ]

    // Таблиця кодів відстаней (0-29)
    distance_codes = [
        {1, 0}, {2, 0}, {3, 0}, {4, 0},
```

```
        {5, 1}, {7, 1}, {9, 2}, {13, 2},
        {17, 3}, {25, 3}, {33, 4}, {49, 4},
        {65, 5}, {97, 5}, {129, 6}, {193, 6},
        {257, 7}, {385, 7}, {513, 8}, {769, 8},
        {1025, 9}, {1537, 9}, {2049, 10}, {3073, 10},
        {4097, 11}, {6145, 11}, {8193, 12}, {12289, 12},
        {16385, 13}, {24577, 13}
    ]

    encoded_data = []

    для кожен елемент в raw_data:
        якщо елемент є Літерал:
            encoded_data.додати(елемент.значення)
        інакше якщо елемент є ПараДовжинаВідстань:
            // Кодуємо довжину
            length_code = знайти_код_довжини(елемент.довжина, length_
codes)
            encoded_data.додати(length_code)

            // Кодуємо відстань
            distance_code = знайти_код_відстані(елемент.відстань, dis
tance_codes)
            encoded_data.додати(distance_code)

    encoded_data.додати(256)  // Маркер кінця блоку

    повернути encoded_data
```

Побудова динамічних дерев Хаффмана

```
Алгоритм BuildDynamicHuffmanTrees(encoded_data):
    Вхід: encoded_data - закодовані дані LZ77
    Вихід: literal_tree, distance_tree - дерева Хаффмана

    // Підрахунок частот для літералів та довжин
    literal_frequencies = новий масив[286] з нулями
    distance_frequencies = новий масив[30] з нулями

    для кожен елемент в encoded_data:
        якщо елемент <= 256 або (257 <= елемент <= 285):
            literal_frequencies[елемент] += 1
        інакше якщо 0 <= елемент <= 29:
            distance_frequencies[елемент] += 1

    // Побудова дерева літералів/довжин
```

```
    literal_tree = BuildHuffmanTree(literal_frequencies, 286)

    // Побудова дерева відстаней
    distance_tree = BuildHuffmanTree(distance_frequencies, 30)

    повернути literal_tree, distance_tree

Алгоритм BuildHuffmanTree(frequencies, alphabet_size):
    // Використовуємо стандартний алгоритм Хаффмана
    // з обмеженням на максимальну довжину коду (15 бітів для DEFLATE
)

    nodes = []

    // Створюємо листки для символів з ненульовими частотами
    для i від 0 до alphabet_size - 1:
        якщо frequencies[i] > 0:
            node = новий Node()
            node.symbol = i
            node.frequency = frequencies[i]
            nodes.додати(node)

    // Будуємо дерево знизу вгору
    heap = новий МінКупа(nodes)

    поки heap.розмір() > 1:
        left = heap.вилучити_мін()
        right = heap.вилучити_мін()

        parent = новий Node()
        parent.frequency = left.frequency + right.frequency
        parent.left = left
        parent.right = right

        heap.вставити(parent)

    root = heap.вилучити_мін()

    // Обмежуємо максимальну довжину коду
    LimitCodeLength(root, 15)

    повернути root
```

Кодування дерев для передачі

Особливістю DEFLATE є те, що структура динамічних дерев також має бути передана декодеру. Для цього використовується третє дерево Хаффмана:

```
Алгоритм EncodeTreeStructure(literal_tree, distance_tree):
    Вхід: literal_tree, distance_tree - дерева Хаффмана
    Вихід: encoded_structure - закодована структура дерев

    // Отримуємо довжини кодів для кожного символу
    literal_lengths = GetCodeLengths(literal_tree, 286)
    distance_lengths = GetCodeLengths(distance_tree, 30)

    // Об'єднуємо довжини кодів
    all_lengths = literal_lengths + distance_lengths

    // Кодуємо послідовність довжин використовуючи RLE
    code_lengths_rle = RunLengthEncode(all_lengths)

    // Створюємо дерево для кодування довжин
    code_length_frequencies = новий масив[19] з нулями

    для кожен (символ, кількість) в code_lengths_rle:
        code_length_frequencies[символ] += 1

    code_length_tree = BuildHuffmanTree(code_length_frequencies, 19)

    повернути code_length_tree, code_lengths_rle

Алгоритм RunLengthEncode(lengths):
    // Спеціальне RLE кодування для DEFLATE
    result = []
    i = 0

    поки i < довжина(lengths):
        поточна_довжина = lengths[i]
        count = 1

        // Рахуємо однакові послідовні довжини
        поки i + count < довжина(lengths) та lengths[i + count] == по
точна_довжина:
            count += 1

        якщо поточна_довжина == 0:
            якщо count >= 3:
                // Кодуємо послідовність нулів
                якщо count <= 10:
```

```
                result.додати((17, count))  // 3-10 нулів
            інакше:
                result.додати((18, count))  // 11-138 нулів
        інакше:
            для j від 0 до count - 1:
                result.додати((0, 1))
    інакше:
        result.додати((поточна_довжина, 1))
        якщо count >= 4:
            // Повторення попередньої довжини
            result.додати((16, count - 1))

    i += count

повернути result
```

Розглянемо конкретний приклад стиснення рядка “ABRACADABRA_ABRACADABRA”.

Крок 1: Аналіз LZ77

Позиція	Символ	Вікно пошуку	Результат
0	A	[]	Літерал(A)
1	B	[A]	Літерал(B)
2	R	[AB]	Літерал(R)
3	A	[ABR]	Літерал(A)
4	C	[ABRA]	Літерал(C)
5	A	[ABRAC]	Літерал(A)
6	D	[ABRACA]	Літерал(D)
7	A	[ABRACAD]	Літерал(A)
8	B	[ABRACADA]	Літерал(B)
9	R	[ABRACADAB]	Літерал(R)
10	A	[ABRACADABR]	Літерал(A)
11	_	[ABRACADABRA]	Літерал(_)
12	A	[ABRACADABRA_]	Пара(11, 12)

Результат LZ77: [A, B, R, A, C, A, D, A, B, R, A, _, (11, 12)]

Крок 2: Кодування довжин та відстаней

Пара (11, 12) означає: - Довжина 11 → код 265 (базова довжина 11, 0 додаткових бітів) - Відстань 12 → код 5 (базова відстань 13, потребує 2 додаткові біти)

Крок 3: Підрахунок частот

Частоти літералів: - A (65): 6 разів - B (66): 2 рази - R (82): 2 рази - C (67): 1 раз - D (68): 1 раз - _ (95): 1 раз - 256 (кінець): 1 раз - 265 (довжина 11): 1 раз

Частоти відстаней: - 5 (відстань 12): 1 раз

Крок 4: Побудова дерева літералів/довжин

Для кращого розуміння процесу побудови дерева Хаффмана, розглянемо детально кожен крок об’єднання вузлів.

Початковий стан: Маємо 8 вузлів-листків з частотами: - C(67): 1 - D(68): 1 - _(95): 1 - 256: 1 - 265: 1 - B(66): 2 - R(82): 2 - A(65): 6

Вузли поміщені в мін-купу, впорядковану за частотами.

Ітерація 1: Вилучаємо два вузли з найменшими частотами: C(1) та D(1)

```
Мін-купа: [C(1), D(1), _(1), 256(1), 265(1), B(2), R(2), A(6)]
Вилучаємо: C(1), D(1)
Створюємо: CD(2)

      CD(2)
     /    \
    /      \
  C(1)    D(1)

Додаємо CD(2) назад до купи
Мін-купа: [_(1), 256(1), 265(1), B(2), R(2), CD(2), A(6)]
```

Ітерація 2: Вилучаємо _(1) та 256(1)

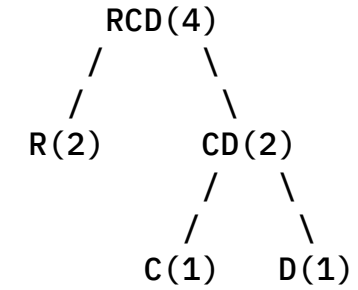
```
Мін-купа: [_(1), 256(1), 265(1), B(2), R(2), CD(2), A(6)]
Вилучаємо: _(1), 256(1)
Створюємо: _256(2)

     _256(2)
     /    \
    /      \
  _(1)   256(1)

Мін-купа: [265(1), B(2), R(2), CD(2), _256(2), A(6)]
```

Ітерація 3: Вилучаємо 265(1) та B(2)

```
Мін-купа: [265(1), B(2), R(2), CD(2), _256(2), A(6)]
Вилучаємо: 265(1), B(2)
Створюємо: 265B(3)

     265B(3)
     /    \
    /      \
  265(1)  B(2)
```

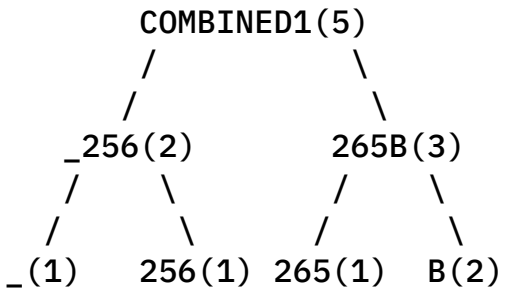
```
Мін-купа: [R(2), CD(2), _256(2), 265B(3), A(6)]
```
Ітерація 4: Вилучаємо R(2) та CD(2)

```
Мін-купа: [R(2), CD(2), _256(2), 265B(3), A(6)]
Вилучаємо: R(2), CD(2)
Створюємо: RCD(4)
```



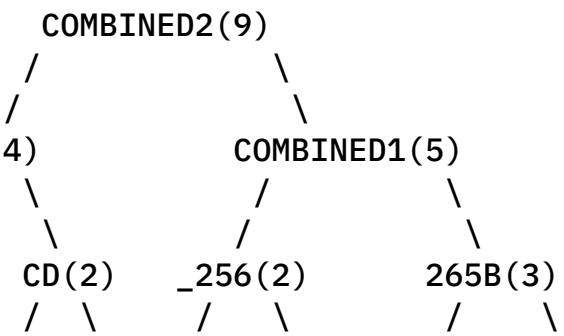

```
Мін-купа: [_256(2), 265B(3), RCD(4), A(6)]
```
Ітерація 5: Вилучаємо _256(2) та 265B(3)

```
Мін-купа: [_256(2), 265B(3), RCD(4), A(6)]
Вилучаємо: _256(2), 265B(3)
Створюємо: COMBINED1(5)
```




```
Мін-купа: [RCD(4), COMBINED1(5), A(6)]
```
Ітерація 6: Вилучаємо RCD(4) та COMBINED1(5)

```
Мін-купа: [RCD(4), COMBINED1(5), A(6)]
Вилучаємо: RCD(4), COMBINED1(5)
Створюємо: COMBINED2(9)
```

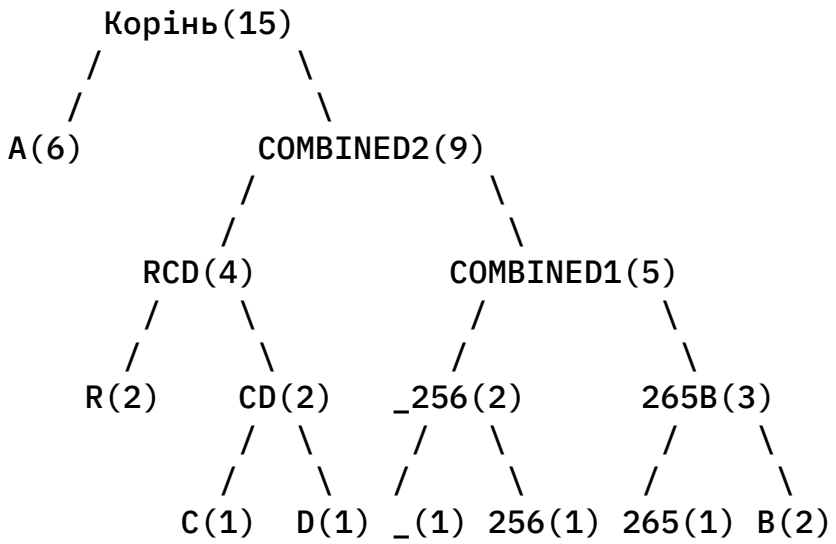


```
  /    \   /     \      /      \
C(1)  D(1) _(1) 256(1) 265(1) B(2)
```

```
Мін-купа: [A(6), COMBINED2(9)]
```

Ітерація 7 (фінальна): Вилучаємо A(6) та COMBINED2(9)

```
Мін-купа: [A(6), COMBINED2(9)]
Вилучаємо: A(6), COMBINED2(9)
Створюємо: Корінь(15)
```



Визначення кодів з побудованого дерева:

Тепер проходимо від кореня до кожного листка, записуючи шлях: - Ліворуч = 0 - Праворуч = 1

```
A:   0 (корінь → ліворуч)
R:   100 (корінь → праворуч → ліворуч → ліворуч)
C:   10100 (корінь → праворуч → ліворуч → праворуч → ліворуч)
D:   10101 (корінь → праворуч → ліворуч → праворуч → праворуч)
_:   1100 (корінь → праворуч → праворуч → ліворуч → ліворуч)
256: 1101 (корінь → праворуч → праворуч → ліворуч → праворуч)
265: 1110 (корінь → праворуч → праворуч → праворуч → ліворуч)
B:   1111 (корінь → праворуч → праворуч → праворуч → праворуч)
```

Аналіз отриманих кодів:

Зверніть увагу на важливі властивості отриманого коду:

1. Оптимальність довжин: Символ A з найвищою частотою (6) отримав найкоротший код (1 біт). Символи з найменшими частотами (C, D) отримали найдовші коди (5 бітів).

2. Префіксність: Жоден код не є префіксом іншого. Наприклад, код “0” (для A) не є початком жодного іншого коду.

3. Збалансованість: Дерево намагається залишатися відносно збалансованим, що забезпечує ефективне декодування.

Обчислення середньої довжини коду:

Середня довжина = Σ(частота × довжина коду) / Σ(частота)

```
A: 6 × 1 = 6 бітів
B: 2 × 4 = 8 бітів
R: 2 × 3 = 6 бітів
C: 1 × 5 = 5 бітів
D: 1 × 5 = 5 бітів
_: 1 × 4 = 4 біти
256: 1 × 4 = 4 біти
265: 1 × 4 = 4 біти

Всього: 42 біти для 15 символів
Середня довжина: 42 / 15 = 2.8 біти/символ
```

Порівняно з фіксованим кодуванням (3 біти для 8 символів), ми досягли покращення.

Крок 5: Побудова дерева відстаней

Для нашого прикладу є лише одна відстань: - Код 5: частота 1

Дерево містить лише один вузол, код: 0

Фінальне кодування даних

```
Алгоритм DEFLATEEncode(input_data):
    Вхід: input_data - вхідні дані
    Вихід: compressed_data - стиснені дані

    // Етап 1: LZ77 стиснення
    lz77_output = LZ77_Compress(input_data, 32768, 258)

    // Етап 2: Кодування довжин та відстаней
    encoded_data = EncodeLengthDistance(lz77_output)

    // Етап 3: Побудова дерев Хаффмана
    literal_tree, distance_tree = BuildDynamicHuffmanTrees(encoded_da
ta)

    // Етап 4: Кодування структури дерев
    tree_structure = EncodeTreeStructure(literal_tree, distance_tree)

    // Етап 5: Кодування даних
    compressed_stream = []
```

```
    // Заголовок блоку (3 біти)
    compressed_stream.додати_біти(1, 1)  // Останній блок
    compressed_stream.додати_біти(2, 2)  // Динамічні коди Хаффмана

    // Додаємо закодовану структуру дерев
    compressed_stream.додати(tree_structure)

    // Кодуємо дані використовуючи побудовані дерева
    для кожен елемент в encoded_data:
        якщо елемент <= 285:  // Літерал або довжина
            код = GetHuffmanCode(literal_tree, елемент)
            compressed_stream.додати_біти(код)

            якщо 257 <= елемент <= 285:
                // Додаємо додаткові біти для довжини
                extra_bits = GetLengthExtraBits(елемент)
                compressed_stream.додати_біти(extra_bits)
        інакше:  // Відстань
            код = GetHuffmanCode(distance_tree, елемент)
            compressed_stream.додати_біти(код)

            // Додаємо додаткові біти для відстані
            extra_bits = GetDistanceExtraBits(елемент)
            compressed_stream.додати_біти(extra_bits)

    повернути compressed_stream
```

Алгоритм декодування DEFLATE

```
Алгоритм DEFLATEDecode(compressed_data):
    Вхід: compressed_data - стиснені дані
    Вихід: decompressed_data - відновлені дані

    decompressed_data = []
    stream_position = 0

    поки stream_position < довжина(compressed_data):
        // Читаємо заголовок блоку
        is_final = читати_біти(compressed_data, stream_position, 1)
        block_type = читати_біти(compressed_data, stream_position, 2)

        якщо block_type == 0:  // Нестиснений блок
            DecodeUncompressedBlock(compressed_data, stream_position,
decompressed_data)
        інакше якщо block_type == 1:  // Фіксовані коди
```

```
            DecodeFixedHuffmanBlock(compressed_data, stream_position,
decompressed_data)
        інакше якщо block_type == 2:  // Динамічні коди
            DecodeDynamicHuffmanBlock(compressed_data, stream_positio
n, decompressed_data)

        якщо is_final:
            перервати

    повернути decompressed_data

Процедура DecodeDynamicHuffmanBlock(compressed_data, position, output
):
    // Читаємо параметри дерев
    hlit = читати_біти(5) + 257  // Кількість кодів літералів/довжин
    hdist = читати_біти(5) + 1   // Кількість кодів відстаней
    hclen = читати_біти(4) + 4   // Кількість кодів довжин кодів

    // Відновлюємо дерево кодів довжин
    code_length_tree = DecodeCodeLengthTree(compressed_data, position
, hclen)

    // Відновлюємо основні дерева
    literal_tree = DecodeTree(compressed_data, position, hlit, code_l
ength_tree)
    distance_tree = DecodeTree(compressed_data, position, hdist, code
_length_tree)

    // Декодуємо дані
    поки true:
        символ = DecodeSymbol(compressed_data, position, literal_tree
)

        якщо символ < 256:  // Літерал
            output.додати(символ)
        інакше якщо символ == 256:  // Кінець блоку
            перервати
        інакше:  // Пара довжина-відстань
            довжина = DecodeLenth(символ, compressed_data, position)
            код_відстані = DecodeSymbol(compressed_data, position, di
stance_tree)
            відстань = DecodeDistance(код_відстані, compressed_data,
position)

            // Копіюємо дані з буфера
```

```
            для і від 0 до довжина - 1:
                output.додати(output[довжина(output) - відстань])
```

Область застосування

Алгоритм DEFLATE є одним з найпоширеніших алгоритмів стиснення даних і використовується в багатьох сферах:

- Архіватори файлів: DEFLATE є основою форматів ZIP, GZIP, та багатьох інших. Практично всі сучасні операційні системи мають вбудовану підтримку цих форматів.

- Веб-технології: HTTP протокол підтримує стиснення контенту за допомогою GZIP (який використовує DEFLATE). Це дозволяє значно зменшити обсяг даних, що передаються між сервером та клієнтом.

- Формати зображень: PNG формат використовує DEFLATE для стиснення растрових даних після фільтрації. Це забезпечує стиснення без втрат для зображень.

- Документи: Формати Office Open XML (DOCX, XLSX, PPTX) та OpenDocument (ODF) використовують ZIP-контейнери з DEFLATE стисненням для зберігання компонентів документів.

- Системи контролю версій: Git використовує DEFLATE для стиснення об’єктів у репозиторії, що дозволяє ефективно зберігати історію змін.

- Мережеві протоколи: Багато протоколів, включаючи SSH та TLS, можуть використовувати DEFLATE для стиснення даних під час передачі.

Приклади застосування

Приклад 1: Стиснення HTML сторінки

Розглянемо типову HTML сторінку розміром 50KB. HTML містить багато повторюваних тегів та атрибутів:

```
<div class="container">
  <div class="header">
    <div class="navigation">
      ...
    </div>
  </div>
  <div class="content">
    ...
  </div>
</div>
```

LZ77 знайде повтори: - `<div class="` зустрічається десятки разів - `</div>` повторюється ще частіше - Атрибути як `class=`, `id=`, `href=` є частими

Результат: типове стиснення 70-80%, фінальний розмір 10-15KB.

Приклад 2: Стиснення JSON даних

JSON файл з даними API:

```
{
  "users": [
    {"id": 1, "name": "John", "email": "john@example.com", "role": "admin"},
    {"id": 2, "name": "Jane", "email": "jane@example.com", "role": "user"},
    {"id": 3, "name": "Bob", "email": "bob@example.com", "role": "user"}
  ]
}
```

DEFLATE ефективно стискає: - Повторювані ключі: “id”, “name”, “email”, “role” - Повторювані значення: “user”, “@example.com” - Структурні елементи: фігурні дужки, коми, лапки

Коефіцієнт стиснення: 60-70%.

Приклад 3: Стиснення логів програми

Лог-файл містить багато однотипних записів:

```
2024-01-15 10:23:45 INFO Application started
2024-01-15 10:23:46 DEBUG Loading configuration
2024-01-15 10:23:46 DEBUG Configuration loaded successfully
2024-01-15 10:23:47 INFO Connecting to database
2024-01-15 10:23:48 INFO Connected to database
```

DEFLATE знаходить та ефективно кодує: - Повторювані дати та часові мітки - Рівні логування: INFO, DEBUG - Часті фрази: “Configuration”, “database”

Типовий коефіцієнт стиснення для логів: 80-90%.

Переваги та недоліки

Переваги:

- Високий коефіцієнт стиснення: Комбінація LZ77 та Хаффмана забезпечує відмінне стиснення для більшості типів даних

- Універсальність: Ефективно працює з текстом, бінарними даними, зображеннями

- Швидка декомпресія: Процес декодування простіший за кодування, що важливо для багатьох застосувань

- Стандартизація: RFC 1951 детально описує алгоритм, забезпечуючи сумісність реалізацій

- Потокове стиснення: Можливість обробки даних блоками дозволяє працювати з потоками необмеженого розміру

- Гнучкість: Три типи блоків дозволяють адаптуватися до різних типів даних

Недоліки:

- Обчислювальна складність: Пошук збігів у LZ77 вимагає значних обчислювальних ресурсів

- Використання пам’яті: Вікно розміром 32КВ та додаткові структури даних вимагають помітного обсягу RAM

- Затримка: Необхідність аналізу даних перед стисненням створює затримку, що може бути критично для real-time систем

- Обмеження на розмір збігів: Максимальна довжина збігу 258 байт може обмежувати ефективність для деяких типів даних

- Складність реалізації: Повна оптимізована реалізація DEFLATE є досить складною задачею

- Відсутність спеціалізації: Універсальність алгоритму означає, що для специфічних типів даних існують кращі алгоритми

Контрольні запитання

1. З яких двох основних алгоритмів складається DEFLATE?

2. Які три типи блоків може містити стиснений потік DEFLATE?

3. Яке максимальне значення довжини збігу в алгоритмі LZ77 для DEFLATE?

4. Скільки дерев Хаффмана використовується для кодування одного динамічного блоку?

5. Який розмір вікна пошуку використовується в DEFLATE?

6. Що кодує значення 256 в дереві літералів/довжин?

7. Яка максимальна довжина коду Хаффмана в DEFLATE?

8. Для чого використовуються додаткові біти при кодуванні довжин та відстаней?

9. Як кодується структура динамічних дерев Хаффмана для передачі?

10. Які діапазони значень кодують дерева літералів/довжин та відстаней?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму DEFLATE для ефективного стиснення даних

2. Мета

Вивчити принципи комбінованого стиснення даних, освоїти роботу з алгоритмами LZ77 та Хаффмана, реалізувати повний цикл стиснення та декомпресії за стандартом DEFLATE

3. Завдання

- Реалізувати алгоритм LZ77 з ковзним вікном
- Створити систему кодування довжин та відстаней
- Реалізувати побудову динамічних дерев Хаффмана
- Створити модуль кодування структури дерев
- Реалізувати формування стиснених блоків
- Створити декодер DEFLATE
- Реалізувати підтримку всіх трьох типів блоків
- Провести тестування та оптимізацію

4. Опис

Лабораторна робота передбачає створення повної реалізації алгоритму DEFLATE, сумісної зі стандартом RFC 1951. Програма повинна ефективно стискати різні типи даних, підтримувати потокову обробку та забезпечувати коректну декомпресію.

5. Порядок виконання

- Реалізувати структури даних для вікна пошуку та буфера передогляду
- Створити ефективний алгоритм пошуку збігів (хеш-таблиця або суфіксне дерево)
- Реалізувати кодування пар довжина-відстань згідно специфікації
- Створити модуль побудови обмежених дерев Хаффмана
- Реалізувати RLE кодування для структури дерев
- Створити бітовий потік для запису стиснених даних
- Реалізувати формування заголовків блоків
- Створити декодер з підтримкою всіх типів блоків
- Додати CRC32 контрольну суму для перевірки цілісності
- Оптимізувати продуктивність критичних частин алгоритму

6. Результат

Бібліотека стиснення даних, сумісна з форматами ZIP/GZIP, з документацією, прикладами використання та порівняльним аналізом ефективності

Після виконання лабораторної роботи студент повинен надати:

- **Вихідний код програми, де чітко виділені:**
- Модуль LZ77 з хеш-таблицею для швидкого пошуку
- Структури даних для вікна та буфера
- Модуль кодування довжин та відстаней
- Алгоритм побудови дерев Хаффмана з обмеженням довжини
- Модуль RLE кодування структури дерев
- Бітовий потік з буферизацією
- Формувач блоків DEFLATE
- Декодер з підтримкою всіх типів блоків
- Модуль CRC32 для контролю цілісності
- Інтерфейс для роботи з файлами та потоками
- **Продемонструвати роботу додатку:**
- Стиснення та декомпресія різних типів файлів
- Порівняння з еталонними реалізаціями (gzip, zlib)
- Візуалізація процесу пошуку збігів LZ77
- Відображення статистики використання кодів
- Аналіз розподілу типів блоків для різних даних
- Вимірювання швидкості стиснення/декомпресії
- Демонстрація роботи з потоками даних
- Перевірка сумісності з існуючими архіваторами
- **Бути готовим відповісти на питання:**
- Як оптимізувати пошук збігів у LZ77?
- Чому використовуються два окремі дерева Хаффмана?
- Як вибирається тип блоку для конкретних даних?
- Які компроміси між швидкістю та ступенем стиснення?
- Як забезпечується сумісність з RFC 1951?

- Які особливості реалізації для різних платформ?

Тести

1. Алгоритм DEFLATE комбінує:

a) LZ77 та арифметичне кодування

b) LZ78 та кодування Хаффмана

c) LZ77 та кодування Хаффмана

d) RLE та кодування Фано

2. Максимальний розмір вікна пошуку в DEFLATE:

a) 8 КВ

b) 16 КВ

c) 32 КВ

d) 64 КВ

3. Значення 256 в дереві літералів/довжин означає:

a) Початок нового блоку

b) Кінець стисненого блоку

c) Максимальний літерал

d) Помилку декодування

4. Скільки типів блоків підтримує DEFLATE?

a) 2

b) 3

c) 4

d) 5

5. Максимальна довжина збігу в DEFLATE:

a) 128 байт

b) 255 байт

c) 258 байт

d) 512 байт

Відповіді: 1. c) 2. c) 3. b) 4. b) 5. c)

Тести підвищеної складності

1. Діапазон кодів для довжин збігів у дереві літералів/довжин:

a) 0-255

b) 256-285

c) 257-285

d) 0-285

2. Кількість бітів для представлення коду відстані (без додаткових бітів):

a) Завжди 5 бітів

b) Від 1 до 5 бітів

c) Залежить від дерева Хаффмана

d) Завжди 15 бітів

3. При кодуванні довжини 35 використовується:

a) Код 269 + 0 додаткових бітів

b) Код 269 + 3 додаткові біти

c) Код 271 + 3 додаткові біти

d) Код 271 + 0 додаткових бітів

4. Максимальна кількість кодів у дереві кодів довжин:

a) 16

b) 19

c) 30

d) 32

5. Порядок передачі довжин кодів для дерева кодів довжин:

a) Послідовно від 0 до 18

b) У спеціальному переставленому порядку

c) Лише ненульові значення

d) У зворотному порядку

6. Мінімальна довжина збігу для кодування парою довжина-відстань:

a) 1 байт

b) 2 байти

c) 3 байти

d) 4 байти

7. RLE код 17 у структурі дерева означає:

a) Повторити попередню довжину

b) 3-10 нульових довжин

c) 11-138 нульових довжин

d) Кінець опису дерева

8. Фіксовані коди Хаффмана в DEFLATE використовують:

a) 7 бітів для літералів 0-143

b) 8 бітів для літералів 0-143

c) 9 бітів для літералів 0-143

d) Змінну довжину для всіх літералів

9. При стисненні вже стиснених даних DEFLATE:

a) Завжди погіршує результат

b) Може використати нестиснені блоки

c) Автоматично пропускає стиснення

d) Видає помилку

10. Теоретична межа стиснення DEFLATE визначається:

a) Лише ентропією джерела

b) Розміром вікна LZ77 та ентропією залишків

c) Типом використаних блоків

d) Швидкістю процесора

Відповіді:

1. c) Коди 257-285 використовуються для представлення довжин збігів від 3 до 258

2. c) Довжина коду відстані визначається динамічним деревом Хаффмана

3. c) Довжина 35 кодується кодом 271 (базова довжина 35) з 3 додатковими бітами

4. b) Дерево кодів довжин може містити до 19 різних символів

5. b) Довжини передаються в порядку: 16,17,18,0,8,7,9,6,10,5,11,4,12,3,13,2,14,1,15

6. c) Мінімальна довжина для ефективного кодування - 3 байти

7. b) Код 17 кодує від 3 до 10 нульових довжин з 3 додатковими бітами

8. b) Фіксована схема використовує 8 бітів для літералів 0-143

9. b) DEFLATE може визначити неефективність стиснення та використати нестиснені блоки

10. b) Ефективність обмежена як розміром вікна для знаходження повторів, так і ентропією того, що залишається

Завдання для самостійної роботи

1. Реалізуйте оптимізований пошук збігів для LZ77 з використанням суфіксного масиву. Порівняйте продуктивність з хеш-таблицею для різних типів даних.

2. Створіть адаптивний вибір типу блоку, який автоматично визначає найефективніший тип (нестиснений, фіксований, динамічний) для кожної порції даних.

3. Розробіть паралельну версію DEFLATE, яка розділяє вхідні дані на незалежні блоки для обробки на багатоядерних системах.

4. Реалізуйте модифікацію DEFLATE з розширеним вікном (64KB або 128KB) та дослідіть вплив на коефіцієнт стиснення та продуктивність.

5. Створіть спеціалізовану версію DEFLATE для стиснення геномних даних з урахуванням особливостей ДНК послідовностей.

6. Розробіть інструмент аналізу DEFLATE потоків, який візуалізує структуру блоків, ефективність LZ77 та розподіл кодів Хаффмана.

7. Реалізуйте DEFLATE з підтримкою словника попереднього наповнення для покращення стиснення малих файлів схожого типу.

8. Створіть гібридний алгоритм, який комбінує DEFLATE з BWT (Burrows-Wheeler Transform) для покращення стиснення текстових даних.

9. Розробіть апаратний прискорювач для DEFLATE декомпресії з використанням FPGA або GPU.

10. Дослідіть та реалізуйте методи покращення стійкості DEFLATE до помилок передачі з мінімальним впливом на коефіцієнт стиснення.

Теми лабораторних робіт

1. Розробка високопродуктивної бібліотеки DEFLATE з підтримкою SIMD інструкцій для прискорення пошуку збігів та обчислення CRC.

2. Створення веб-сервера з вбудованою підтримкою DEFLATE стиснення, адаптивним вибором рівня стиснення залежно від навантаження.

3. Реалізація файлової системи з прозорим DEFLATE стисненням, яка автоматично стискає файли при записі та розпаковує при читанні.

4. Розробка інструменту міграції даних між хмарними сховищами з потоковим DEFLATE стисненням для мінімізації трафіку.

5. Створення системи резервного копіювання з диференційним DEFLATE стисненням та дедуплікацією на рівні блоків.

6. Реалізація проксі-сервера для мобільних пристроїв, який стискає весь HTTP трафік використовуючи оптимізований DEFLATE.

7. Розробка плагіна для СУБД, який забезпечує прозоре DEFLATE стиснення BLOB полів з інтелектуальним вибором параметрів.

8. Створення системи віддаленого робочого столу з DEFLATE стисненням графічних даних та адаптацією до пропускної здатності каналу.

9. Реалізація distributed computing фреймворку, який використовує DEFLATE для ефективної передачі завдань та результатів між вузлами.

10. Розробка навчального середовища для вивчення алгоритмів стиснення з інтерактивною візуалізацією кожного етапу роботи DEFLATE.

Розділ 11: Обчислювальна геометрія

Обчислювальна геометрія — це захоплива галузь інформатики, що вивчає методи для вирішення задач із геометричними об’єктами. Її застосування всюдисущі: від розробки відеоігор та систем автоматизованого проєктування (САПР) до геоінформаційних систем (ГІС), робототехніки та комп’ютерного зору.

Ця галузь виникла на перетині класичної геометрії, комп’ютерних наук та дискретної математики, і сьогодні є однією з найважливіших областей теоретичної та прикладної інформатики. Якщо традиційна геометрія займається вивченням властивостей геометричних об’єктів та доведенням теорем, то обчислювальна геометрія фокусується на розробці ефективних алгоритмів для обчислення цих властивостей та розв’язання практичних задач.

Історично, потреба в обчислювальній геометрії виникла в 1970-х роках з розвитком комп’ютерної графіки, систем автоматизованого проектування (CAD) та робототехніки. Піонерські роботи Майкла Шеймоса, Франко Препарати та інших дослідників заклали фундамент цієї дисципліни, визначивши основні класи задач та підходи до їх розв’язання. З того часу обчислювальна геометрія стала невід’ємною частиною багатьох технологій, які ми використовуємо щодня.

Фундаментальні концепції

В основі обчислювальної геометрії лежить представлення геометричних об’єктів у вигляді дискретних структур даних. Точки представляються координатами, лінії — рівняннями або парами точок, багатокутники — послідовностями вершин. Ця дискретизація дозволяє застосовувати алгоритмічні методи до геометричних задач, але водночас породжує унікальні виклики, пов’язані з точністю обчислень, виродженими випадками та ефективністю.

Однією з ключових особливостей обчислювальної геометрії є важливість аналізу складності алгоритмів не лише в середньому випадку, але й у найгіршому випадку. Геометричні дані часто мають специфічну структуру, яка може суттєво впливати на продуктивність алгоритмів. Наприклад, точки можуть бути розподілені рівномірно або згруповані в кластери, лінії можуть бути паралельними або перетинатися в одній точці — всі ці особливості потребують уважного врахування при розробці алгоритмів.

Основні класи задач

Задачі обчислювальної геометрії можна умовно розділити на кілька основних класів, кожен з яких має свої характерні особливості та методи розв'язання:

Задачі побудови включають створення геометричних структур з заданих примітивів. Класичним прикладом є побудова опуклої оболонки — найменшого опуклого багатокутника, що містить задану множину точок. Ця задача має численні застосування: від обробки зображень до оптимізації маршрутів. У розділі 11.1 ми детально розглянемо три фундаментальні алгоритми:

Graham scan з його елегантним використанням стеку та кутового сортування,

Jarvis march з інтуїтивно зрозумілим підходом "загортання подарунка",

QuickHull — алгоритм, що використовує стратегію "розділяй і володарюй".

Задачі близькості та сусідства досліджують взаємне розташування геометричних об'єктів. Пошук найближчої пари точок — фундаментальна задача, що має застосування від кластеризації даних до виявлення колізій. Діаграма Вороного та тріангуляція Делоне, які ми вивчатимемо в розділі 11.2, є потужними інструментами для аналізу просторових відношень. Діаграма Вороного розбиває площину на області, кожна з яких містить точки, найближчі до певного центру, що використовується в метеорології, екології та плануванні розміщення об'єктів інфраструктури.

Задачі перетину та належності визначають взаємодію між геометричними об'єктами. Алгоритм Бентлі-Оттмана для знаходження всіх перетинів у множині відрізків є прикладом використання техніки заметання площини — потужного підходу, що дозволяє звести двовимірну задачу до послідовності одновимірних. Задача визначення належності точки багатокутнику, яку ми розв'яжемо методом променевого сканування, є базовою для комп'ютерної графіки, геоінформаційних систем та обробки запитів у просторових базах даних.

Геометричні структури даних, представлені в розділі 11.4, є основою для ефективного розв'язання складних геометричних задач. KD-дерево дозволяє ефективно організувати точки в багатовимірному просторі для швидкого пошуку найближчих сусідів. R-дерево оптимізоване для зберігання та пошуку просторових об'єктів, що робить його незамінним у геоінформаційних системах. BSP-дерево (Binary Space Partitioning)

використовується для розбиття простору гіперплощинами, що критично важливо для рендерингу 3D-графіки та обробки видимості.

Зв'язок з іншими розділами

Обчислювальна геометрія тісно пов'язана з іншими розділами алгоритміки та дискретної математики. Методи "розділяй і володарюй" природно застосовуються до геометричних задач, як це демонструє алгоритм QuickHull. Динамічне програмування використовується для оптимальної тріангуляції багатокутників. Структури даних, такі як збалансовані дерева та черги з пріоритетами, є основою для багатьох геометричних алгоритмів.

Теорія графів також відіграє важливу роль: діаграма Вороного та тріангуляція Делоне можуть розглядатися як спеціальні види планарних графів з унікальними властивостями. Алгоритми на графах часто адаптуються для розв'язання геометричних задач, і навпаки.

11.1. Побудова опуклої оболонки

Уявіть, що на дошці розкидано цвяхи. Якщо взяти гумову стрічку і натягнути її навколо всієї групи цвяхів, форма, яку вона прийме, і є **опуклою оболонкою (convex hull)** . Це одна з найбільш фундаментальних та візуально зрозумілих задач в обчислювальній геометрії.

Формально, опукла оболонка для набору точок на площині — це **найменший опуклий багатокутник**, що містить у собі всі ці точки. Вершинами цього багатокутника будуть деякі з точок початкового набору, які називаються “екстремальними”.

Знаходження опуклої оболонки є не просто цікавою геометричною головоломкою. Це ключовий підготовчий крок для багатьох складніших алгоритмів, що застосовуються в розпізнаванні образів, обробці зображень, моделюванні зіткнень у фізичних рушіях та статистичному аналізі даних.

У цьому розділі ми розглянемо три класичні та елегантні методи її побудови:

- **Сканування Грехема (Graham scan):** Ефективний алгоритм, що сортує точки за полярним кутом і будує оболонку за один прохід.

- **Марш Джарвіса (Jarvis march):** Інтуїтивний метод, схожий на “загортання подарунка”, який послідовно знаходить наступну вершину оболонки.

- **QuickHull:** Алгоритм, ідейно схожий на Quicksort, який рекурсивно відкидає точки, що знаходяться всередині оболонки.

11.1.1. Алгоритм сканування Грема

Теоретичні відомості

Перш ніж розглядати алгоритм сканування Грема, необхідно визначити ключові геометричні поняття, які будуть використовуватися.

Опукла оболонка (convex hull) множини точок на площині — це найменший опуклий багатокутник, який містить всі задані точки. Уявіть, що у вас є набір цвяхів, забитих у дошку, і ви натягуєте навколо них гумку. Форма, яку прийме гумка — це і є опукла оболонка. Математично, точка P належить опуклій оболонці множини S, якщо не існує відрізка між двома точками з S такого, що P лежить строго всередині цього відрізка.

Опуклий багатокутник — це багатокутник, у якому всі внутрішні кути менші за 180 градусів. Еквівалентно, це багатокутник, для якого будь-який відрізок між двома його точками повністю лежить всередині або на межі багатокутника.

Полярний кут точки відносно початку координат — це кут між додатним напрямком осі X та променем, що з’єднує початок координат з даною точкою. Вимірюється проти годинникової стрілки від 0 до 2π радіан.

Поворот (або орієнтація) трьох послідовних точок P, Q, R може бути: - Лівий поворот (counterclockwise) — якщо при русі від P до Q, а потім до R ми повертаємо ліворуч - Правий поворот (clockwise) — якщо повертаємо праворуч - Колінеарні — якщо всі три точки лежать на одній прямій

Для визначення типу повороту використовується векторний добуток. Для трьох точок $P(x_1, y_1)$, $Q(x_2, y_2)$, $R(x_3, y_3)$ обчислюємо визначник:

$$D = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)$$

Інтерпретація результату: - $D > 0$: лівий поворот (проти годинникової стрілки) - $D < 0$: правий поворот (за годинниковою стрілкою) - $D = 0$: точки колінеарні

Ця формула є спрощеною формою векторного добутку векторів PQ та PR. Знак векторного добутку визначає орієнтацію повороту.

Ідея алгоритму сканування Грема

Алгоритм сканування Грема був розроблений Рональдом Гремом у 1972 році для ефективної побудови опуклої оболонки. Основна ідея полягає в тому, щоб:

1. Знайти стартову точку, яка гарантовано належить опуклій оболонці

2. Відсортувати всі інші точки за полярним кутом відносно стартової точки

3. Послідовно обходити відсортовані точки, підтримуючи інваріант опуклості

Алгоритм використовує стек для зберігання потенційних вершин опуклої оболонки та перевіряє, чи утворюють три останні точки лівий поворот. Якщо ні — видаляє передостанню точку зі стеку.

Детальний опис алгоритму

```
Алгоритм GrahamScan(points):
    Вхід: points - масив з n точок на площині
    Вихід: convex_hull - масив точок, що утворюють опуклу оболонку

    // Крок 1: Знайти початкову точку
    p0 = FindLowestPoint(points)

    // Крок 2: Сортування за полярним кутом
    sorted_points = SortByPolarAngle(points, p0)

    // Крок 3: Ініціалізація стеку
    stack = новий стек
    stack.push(p0)
    stack.push(sorted_points[0])
    stack.push(sorted_points[1])

    // Крок 4: Основний цикл сканування
    для i від 2 до n-1:
        // Видаляємо точки, що не утворюють лівий поворот
        поки stack.size() > 1:
            top = stack.top()
            second = stack.second_from_top()

            якщо Orientation(second, top, sorted_points[i]) != COUNTE
RCLOCKWISE:
                stack.pop()
```

```
            інакше:
                перервати

        stack.push(sorted_points[i])

    // Крок 5: Формування результату
    convex_hull = []
    поки не stack.empty():
        convex_hull.додати(stack.pop())

    повернути reverse(convex_hull)
```

Допоміжні функції

```
Функція FindLowestPoint(points):
    Вхід: points - масив точок
    Вихід: точка з найменшою y-координатою

    lowest = points[0]

    для кожної точки p в points:
        якщо p.y < lowest.y або (p.y == lowest.y та p.x < lowest.x):
            lowest = p

    повернути lowest

Функція SortByPolarAngle(points, p0):
    Вхід: points - масив точок, p0 - початкова точка
    Вихід: відсортований масив точок

    // Видаляємо p0 з масиву
    remaining_points = []
    для кожної точки p в points:
        якщо p != p0:
            remaining_points.додати(p)

    // Сортуємо за полярним кутом
    remaining_points.sort(функція порівняння(a, b):
        angle_a = atan2(a.y - p0.y, a.x - p0.x)
        angle_b = atan2(b.y - p0.y, b.x - p0.x)

        якщо angle_a != angle_b:
            повернути angle_a < angle_b
        інакше:
            // При однакових кутах, ближча точка йде першою
            dist_a = Distance(p0, a)
```

```
            dist_b = Distance(p0, b)
            повернути dist_a < dist_b
    )

    повернути remaining_points

Функція Orientation(p, q, r):
    Вхід: три точки p, q, r
    Вихід: тип повороту (LEFT, RIGHT, COLLINEAR)

    val = (q.x - p.x) * (r.y - p.y) - (q.y - p.y) * (r.x - p.x)

    якщо val > 0:
        повернути LEFT  // Лівий поворот
    інакше якщо val < 0:
        повернути RIGHT  // Правий поворот
    інакше:
        повернути COLLINEAR  // Колінеарні

Функція Distance(p1, p2):
    повернути sqrt((p2.x - p1.x)² + (p2.y - p1.y)²)
```

Обробка особливих випадків

При реалізації алгоритму важливо правильно обробляти особливі випадки:

Колінеарні точки при сортуванні: Якщо кілька точок мають однаковий полярний кут відносно p0, їх потрібно впорядкувати за відстанню від p0. Ближчі точки йдуть першими, за винятком останньої групи колінеарних точок — там порядок має бути зворотній.

```
Функція HandleCollinearPoints(sorted_points, p0):
    // Знаходимо точки з максимальним кутом
    max_angle = PolarAngle(sorted_points[n-1], p0)

    // Знаходимо всі точки з цим кутом
    i = n - 1
    поки i >= 0 та PolarAngle(sorted_points[i], p0) == max_angle:
        i = i - 1

    // Реверсуємо порядок цих точок
    reverse(sorted_points, i + 1, n - 1)
```

Вироджені випадки: - Всі точки колінеарні: опукла оболонка — це відрізок між двома крайніми точками - Всі точки однакові: опукла оболонка — це одна точка - Менше трьох різних точок: спеціальна обробка

Розглянемо роботу алгоритму на конкретному прикладі з 8 точками:

```
Початкові точки:
A(2, 3), B(5, 4), C(6, 1), D(7, 4),
E(6, 6), F(3, 5), G(1, 2), H(4, 2)
```

Крок 1: Знаходимо найнижчу точку

```
Перевіряємо y-координати:
A: y=3, B: y=4, C: y=1, D: y=4,
E: y=6, F: y=5, G: y=2, H: y=2

Найменша y-координата: 1 (точка C)
p0 = C(6, 1)
```

Крок 2: Сортуємо за полярним кутом відносно C

```
Обчислюємо полярні кути:
H: atan2(2-1, 4-6) = atan2(1, -2) ≈ 2.68 рад
G: atan2(2-1, 1-6) = atan2(1, -5) ≈ 2.94 рад
A: atan2(3-1, 2-6) = atan2(2, -4) ≈ 2.68 рад
F: atan2(5-1, 3-6) = atan2(4, -3) ≈ 2.21 рад
B: atan2(4-1, 5-6) = atan2(3, -1) ≈ 1.89 рад
E: atan2(6-1, 6-6) = atan2(5, 0) ≈ 1.57 рад
D: atan2(4-1, 7-6) = atan2(3, 1) ≈ 1.25 рад

Відсортований порядок: D, E, B, F, A, H, G
```

Крок 3: Ініціалізація стеку

```
Стек: [C, D, E]
```

Крок 4: Основне сканування

Ітерація 1 (точка B):

```
Перевіряємо поворот D→E→B:
Orientation(D, E, B) = (6-7)*(4-4) - (6-4)*(5-7) = 0 - 2*(-2) = 4 > 0
Лівий поворот → додаємо B
Стек: [C, D, E, B]
```

Ітерація 2 (точка F):

```
Перевіряємо поворот E→B→F:
Orientation(E, B, F) = (5-6)*(5-6) - (4-6)*(3-6) = -1*(-1) - (-2)*(-3
) = 1 - 6 = -5 < 0
Правий поворот → видаляємо B
Стек: [C, D, E]

Перевіряємо поворот D→E→F:
Orientation(D, E, F) = (6-7)*(5-4) - (6-4)*(3-7) = -1*1 - 2*(-4) = -1
+ 8 = 7 > 0
Лівий поворот → додаємо F
Стек: [C, D, E, F]
```

Ітерація 3 (точка A):

```
Перевіряємо поворот E→F→A:
Orientation(E, F, A) = (3-6)*(3-5) - (5-6)*(2-3) = -3*(-2) - (-1)*(-1
) = 6 - 1 = 5 > 0
Лівий поворот → додаємо A
Стек: [C, D, E, F, A]
```

Ітерація 4 (точка H):

```
Перевіряємо поворот F→A→H:
Orientation(F, A, H) = (2-3)*(2-3) - (3-5)*(4-2) = -1*(-1) - (-2)*2 =
1 + 4 = 5 > 0
Лівий поворот → додаємо H
Стек: [C, D, E, F, A, H]
```

Ітерація 5 (точка G):

```
Перевіряємо поворот A→H→G:
Orientation(A, H, G) = (4-2)*(2-2) - (2-3)*(1-4) = 2*0 - (-1)*(-3) =
0 - 3 = -3 < 0
Правий поворот → видаляємо H
Стек: [C, D, E, F, A]

Перевіряємо поворот F→A→G:
Orientation(F, A, G) = (2-3)*(2-3) - (3-5)*(1-2) = -1*(-1) - (-2)*(-1
) = 1 - 2 = -1 < 0
Правий поворот → видаляємо A
Стек: [C, D, E, F]

Перевіряємо поворот E→F→G:
Orientation(E, F, G) = (3-6)*(2-5) - (5-6)*(1-3) = -3*(-3) - (-1)*(-2
) = 9 - 2 = 7 > 0
Лівий поворот → додаємо G
Стек: [C, D, E, F, G]
```

Крок 5: Результат

```
Опукла оболонка: C → D → E → F → G → C
Вершини: C(6,1), D(7,4), E(6,6), F(3,5), G(1,2)
```

Візуалізація алгоритму

```
Початковий стан:
     6 |    E
     5 |  F
     4 |    B D
     3 |  A
     2 |G   H
     1 |      C
     0 +----------
       0123456789
```

```
Після сортування (порядок обходу від C):
C → D → E → B → F → A → H → G

Процес сканування:

Крок 1: [C, D, E]
     6 |    E*
     5 |  F
     4 |    B D*
     3 |  A
     2 |G   H
     1 |      C*

Крок 2: Додаємо B
[C, D, E, B] - лівий поворот

Крок 3: Обробка F
B видаляється (правий поворот E→B→F)
[C, D, E, F] - лівий поворот D→E→F

Крок 4: Додаємо A
[C, D, E, F, A] - лівий поворот

Крок 5: Обробка H
[C, D, E, F, A, H] - лівий поворот

Крок 6: Обробка G
H видаляється (правий поворот A→H→G)
A видаляється (правий поворот F→A→G)
[C, D, E, F, G] - лівий поворот E→F→G

Фінальна опукла оболонка:
     6 |    E*
     5 |  F*
     4 |      D*
     3 |
     2 |G*
     1 |      C*
```

Аналіз складності

Часова складність: - Пошук найнижчої точки: O(n) - Сортування за полярним кутом: O(n log n) - Основне сканування: O(n) - Кожна точка додається в стек максимум один раз - Кожна точка видаляється зі стеку максимум один раз - Загальна кількість операцій: O(n)

Загальна часова складність: O(n log n), де n — кількість точок.

Просторова складність: O(n) для зберігання стеку та відсортованого масиву.

Доведення коректності базується на таких інваріантах:

1. Інваріант стеку: На кожному кроці точки в стеку утворюють опуклий ланцюг від p0 до поточної верхівки стеку

2. Інваріант порядку: Всі точки обробляються в порядку неспадання полярного кута відносно p0

3. Інваріант повноти: Жодна точка опуклої оболонки не буде видалена зі стеку назавжди

Область застосування

Алгоритм сканування Грема широко використовується в різних галузях:

- Комп'ютерна графіка: Визначення меж об'єктів, обчислення тіней, спрощення полігонів, виявлення колізій

- Розпізнавання образів: Виділення контурів об'єктів, класифікація форм, аналіз рукописного тексту

- Географічні інформаційні системи (ГІС): Визначення меж територій, аналіз розподілу об'єктів, планування маршрутів

- Робототехніка: Планування шляху роботів, уникнення перешкод, визначення робочої зони маніпулятора

- Обчислювальна біологія: Аналіз форми білків, моделювання клітинних структур, дослідження розподілу популяцій

- Статистика та аналіз даних: Виявлення викидів, кластеризація даних, візуалізація багатовимірних даних (після проекції)

- Комп'ютерне моделювання: Фізичні симуляції, моделювання рідин, обчислення аеродинамічних характеристик

Приклади застосування

Приклад 1: Система безпеки периметру

Компанія встановлює камери спостереження для охорони території. Маючи координати всіх об'єктів, що потребують охорони, алгоритм сканування Грема визначає мінімальний периметр для розміщення камер.

```
Вхідні дані: координати складів, офісів, парковок
Результат: оптимальний контур для розміщення огорожі та камер
Економія: до 30% на матеріалах порівняно з прямокутним периметром
```

Приклад 2: Оптимізація доставки

Кур’єрська служба використовує опуклу оболонку для визначення зони обслуговування та планування маршрутів.

```
Алгоритм DeliveryZoneOptimization(delivery_points):
    hull = GrahamScan(delivery_points)
    zone_area = CalculateArea(hull)

    якщо zone_area > MAX_ZONE_AREA:
        // Розділити на підзони
        zones = SplitIntoSubzones(delivery_points)
    інакше:
        // Оптимізувати маршрут в межах зони
        route = OptimizeRoute(delivery_points, hull)

    повернути route
```

Приклад 3: Аналіз кластерів у машинному навчанні

При кластеризації даних опукла оболонка допомагає візуалізувати та аналізувати межі кластерів.

```
Функція ClusterBoundaryAnalysis(cluster_points):
    hull = GrahamScan(cluster_points)

    // Характеристики кластера
    perimeter = CalculatePerimeter(hull)
    area = CalculateArea(hull)
    compactness = 4 * π * area / (perimeter²)

    // Визначення форми кластера
    якщо compactness > 0.9:
        shape = "Круглий"
    інакше якщо AspectRatio(hull) > 3:
        shape = "Витягнутий"
    інакше:
        shape = "Неправильний"

    повернути {hull, shape, compactness}
```

Переваги та недоліки

Переваги:

- Оптимальна часова складність: O(n log n) є найкращою можливою для задачі побудови опуклої оболонки в загальному випадку

- Простота реалізації: Алгоритм використовує прості структури даних (стек) та базові геометричні операції

- Стійкість: Завжди знаходить правильну опуклу оболонку, не залежить від розподілу точок

- Ефективність на практиці: Добре працює з реальними даними, має малу константу в O-нотації

- Передбачуваність: Завжди виконується за O(n log n), без найгіршого випадку $O(n^2)$

- Можливість паралелізації: Етап сортування можна розпаралелити

Недоліки:

- Необхідність сортування: Вимагає O(n log n) часу навіть якщо точки вже частково впорядковані

- Чутливість до числових похибок: При обчисленні кутів можуть виникати проблеми з точністю для близьких точок

- Складність узагальнення: Важко адаптувати для тривимірного простору (існують інші алгоритми)

- Використання тригонометрії: Обчислення atan2 може бути повільним, хоча існують способи уникнути цього

- Не онлайновий: Потребує всіх точок заздалегідь, не може ефективно обробляти динамічні додавання точок

- Додаткова пам’ять: Потребує O(n) додаткової пам’яті для сортування та стеку

Контрольні запитання

1. Що таке опукла оболонка множини точок і які її основні властивості?

2. Поясніть математичний принцип визначення типу повороту трьох точок.

3. Чому в алгоритмі сканування Грема використовується саме найнижча точка як початкова?

4. Яка роль стеку в алгоритмі та який інваріант він підтримує?

5. Як обробляються колінеарні точки при сортуванні за полярним кутом?

6. Доведіть, що кожна точка додається та видаляється зі стеку максимум один раз.

7. Чому часова складність алгоритму становить O(n log n)?

8. Які модифікації алгоритму можливі для уникнення обчислення тригонометричних функцій?

9. Як адаптувати алгоритм для випадку, коли потрібна опукла оболонка без колінеарних точок на межі?

10. Порівняйте алгоритм сканування Грема з іншими алгоритмами побудови опуклої оболонки.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму сканування Грема для побудови опуклої оболонки

2. Мета

Вивчити принципи роботи алгоритму сканування Грема, навчитися реалізовувати геометричні алгоритми, освоїти техніки роботи з полярними координатами та обробки геометричних даних

3. Завдання

- Реалізувати функцію визначення орієнтації трьох точок

- Реалізувати сортування точок за полярним кутом

- Створити основну функцію алгоритму сканування Грема

- Реалізувати візуалізацію процесу побудови опуклої оболонки

- Додати обробку особливих випадків (колінеарні точки, дублікати)

- Провести тестування на різних наборах даних

- Порівняти продуктивність з наївним алгоритмом

4. Опис

Лабораторна робота передбачає повну реалізацію алгоритму сканування Грема з візуалізацією кожного кроку. Програма повинна коректно обробляти різні конфігурації точок, включаючи вироджені випадки, та надавати можливість покрокового виконання алгоритму для навчальних цілей.

5. Порядок виконання

- Створити клас Point для представлення точок на площині

- Реалізувати допоміжні геометричні функції (відстань, орієнтація)

- Створити функцію пошуку початкової точки

- Реалізувати компаратор для сортування за полярним кутом

- Написати основну функцію алгоритму зі стеком
- Додати модуль візуалізації з використанням графічної бібліотеки
- Створити генератор тестових даних (випадкові, на колі, кластери)
- Реалізувати покрокове виконання з візуалізацією стану стеку
- Додати можливість завантаження точок з файлу
- Провести тестування та оптимізацію

6. Результат

Програма з графічним інтерфейсом, яка дозволяє вводити точки, виконувати алгоритм сканування Грема з візуалізацією, зберігати та завантажувати набори точок, а також порівнювати результати з іншими алгоритмами побудови опуклої оболонки.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
- Клас Point з перевантаженими операторами
- Функція обчислення орієнтації трьох точок
- Функція сортування за полярним кутом без використання atan2
- Основний алгоритм сканування Грема
- Модуль обробки особливих випадків
- Модуль візуалізації з анімацією
- Генератор тестових наборів точок
- Модуль порівняння алгоритмів
- Продемонструвати роботу додатку:
- Введення точок мишкою та з клавіатури
- Покрокове виконання алгоритму з поясненнями
- Візуалізація стану стеку на кожному кроці
- Обробка різних тестових наборів (випадкові, на колі, сітка)
- Коректна робота з колінеарними точками
- Порівняння часу виконання для різних розмірів даних
- Збереження та завантаження наборів точок

Бути готовим відповісти на питання:

- Як уникнути використання тригонометричних функцій при сортуванні?
- Чому важливо правильно обробляти колінеарні точки?
- Як модифікувати алгоритм для побудови опуклої оболонки онлайн?
- Які існують альтернативні алгоритми та їх переваги?
- Як забезпечити стійкість до числових похибок?
- Як оптимізувати алгоритм для спеціальних випадків розподілу точок?

Тести

1. Яка часова складність алгоритму сканування Грема?

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) $O(n^3)$

2. Яка точка обирається як початкова в алгоритмі сканування Грема?

1) Точка з найменшою x-координатою

2) Точка з найбільшою y-координатою

3) Точка з найменшою y-координатою

4) Випадкова точка

3. Що зберігається в стеку під час роботи алгоритму?

1) Всі оброблені точки

2) Точки потенційної опуклої оболонки

3) Відкинуті точки

4) Точки в порядку обробки

4. Коли точка видаляється зі стеку?

1) Коли вона має найбільшу відстань від початкової точки

2) Коли три останні точки утворюють правий поворот

3) Коли вона колінеарна з сусідніми точками

4) Коли вона дублюється

5. Як визначається орієнтація трьох точок P, Q, R?

1) За допомогою скалярного добутку

2) За допомогою векторного добутку

3) За допомогою відстаней між точками

4) За допомогою кутів між векторами

Відповіді тести:

1. 2)

2. 3)

3. 2)

4. 2)

5. 2)

Тести підвищеної складності

1. Як правильно обробити набір точок, де всі точки мають однаковий полярний кут відносно початкової точки?

1) Відкинути всі крім найближчої точки

2) Включити всі точки в опуклу оболонку

3) Впорядкувати за відстанню, включити найближчу та найдальшу

4) Випадково вибрати одну точку

2. Яка структура даних найкраще підходить для оптимізації пошуку початкової точки, якщо потрібно будувати опуклі оболонки для багатьох підмножин точок?

1) Збалансоване бінарне дерево пошуку

2) Heap з пріоритетом за y-координатою

3) Хеш-таблиця

4) Масив, відсортований за y-координатою

3. Скільки разів точка може бути додана до стеку протягом виконання алгоритму?

1) Максимум 1 раз

2) Максимум 2 рази

3) Максимум n-1 разів

4) Необмежену кількість разів

4. Що станеться, якщо не обробити правильно колінеарні точки з максимальним полярним кутом?

1) Алгоритм зациклиться

2) Опукла оболонка буде неповною

3) Опукла оболонка буде неправильною

4) Виникне виключення при роботі зі стеком

5. Як можна модифікувати алгоритм для побудови опуклої оболонки без внутрішніх колінеарних точок на межі?

1) Видалити всі колінеарні точки перед сортуванням

2) Змінити умову в основному циклі на нестрогу нерівність

3) Використати інший алгоритм

4) Додати постобробку для видалення колінеарних точок

6. Яка мінімальна кількість точок потрібна для роботи стандартного алгоритму сканування Грема?

1) 1

2) 2

3) 3

4) 4

7. Як уникнути використання функції atan2 при сортуванні за полярним кутом?

1) Використати тільки x-координати

2) Порівнювати векторні добутки

3) Використати наближені значення кутів

4) Сортувати за відстанню

8. Що є інваріантом основного циклу алгоритму?

1) Стек містить всі оброблені точки

2) Точки в стеку утворюють опуклий ланцюг

3) Стек містить фіксовану кількість точок

4) Всі точки в стеку мають різні полярні кути

9. Яка модифікація потрібна для обробки точок з цілочисельними координатами без втрати точності?

1) Використовувати арифметику з плаваючою точкою

2) Масштабувати всі координати

3) Використовувати лише цілочисельні операції для визначення орієнтації

4) Округлювати результати обчислень

10. Як алгоритм сканування Грема можна адаптувати для динамічної побудови опуклої оболонки (з додаванням нових точок)?

1) Перезапускати алгоритм для кожної нової точки

2) Використати інкрементальний алгоритм замість сканування Грема

3) Додавати нові точки в кінець відсортованого масиву

4) Зберігати стек між викликами

Відповіді:

1. 3) При однакових кутах включаємо найближчу та найдальшу точки, проміжні колінеарні точки не входять в опуклу оболонку

2. 2) Heap дозволяє за O(1) отримати точку з мінімальною y-координатою та за O(log n) оновлювати структуру

3. 1) Кожна точка додається до стеку максимум один раз під час основного сканування

4. 2) Без правильної обробки останніх колінеарних точок алгоритм може пропустити частину опуклої оболонки

5. 2) Змінивши строгу нерівність на нестрогу при перевірці орієнтації, колінеарні точки залишаться в оболонці

6. 3) Потрібно мінімум 3 точки для ініціалізації стеку та початку основного циклу

7. 2) Порівняння векторних добутків дозволяє визначити порядок без обчислення кутів

8. 2) Точки в стеку завжди утворюють опуклий ланцюг від початкової точки до верхівки

9. 3) Використання лише цілочисельної арифметики в формулі орієнтації гарантує точність

10. 2) Для динамічної побудови краще використати інкрементальні алгоритми, оскільки сканування Грема вимагає повного пересортування

Завдання для самостійної роботи

1. Реалізуйте модифікацію алгоритму сканування Грема, яка використовує цілочисельну арифметику для точного обчислення орієнтації точок з цілими координатами.

2. Розробіть паралельну версію алгоритму сканування Грема, розділивши точки на сектори та обробляючи їх незалежно з подальшим об’єднанням результатів.

3. Створіть адаптивний алгоритм, який автоматично вибирає між скануванням Грема та іншими алгоритмами залежно від розподілу точок (наприклад, Jarvis march для малої кількості точок на оболонці).

4. Реалізуйте алгоритм побудови опуклих шарів (convex layers) — послідовне видалення опуклих оболонок до вичерпання всіх точок.

5. Розробіть модифікацію для побудови α-опуклої оболонки (alpha-hull) — узагальнення опуклої оболонки з параметром кривизни.

6. Створіть інтерактивну навчальну програму, яка демонструє покрокове виконання алгоритму з поясненнями кожного рішення та можливістю “відкату” кроків.

7. Реалізуйте алгоритм для знаходження опуклої оболонки точок на сфері, адаптувавши ідеї сканування Грема для сферичної геометрії.

8. Розробіть систему для виявлення та класифікації опуклих об’єктів на зображенні з використанням алгоритму сканування Грема.

9. Створіть оптимізовану версію алгоритму для випадку, коли точки надходять з потоку даних і потрібно підтримувати поточну опуклу оболонку.

10. Реалізуйте застосування алгоритму сканування Грема для задачі пакування: знаходження мінімального опуклого контейнера для набору багатокутників.

Теми лабораторних робіт

1. Розробка бібліотеки геометричних алгоритмів з реалізацією різних методів побудови опуклої оболонки та їх порівняльним аналізом.

2. Створення системи аналізу форм для розпізнавання рукописних символів з використанням опуклої оболонки як однієї з ознак.

3. Реалізація геометричного ядра для 2D CAD системи з підтримкою побудови опуклих оболонок складних об’єктів.

4. Розробка плагіна для ГІС-системи, який використовує опуклі оболонки для аналізу кластерів географічних об’єктів.

5. Створення гри-головоломки, де гравець повинен визначати опуклі оболонки різних наборів точок на швидкість.

6. Реалізація системи планування маршрутів для роботів з використанням опуклих оболонок для представлення перешкод.

7. Розробка інструменту для оптимізації розміщення базових станцій мобільного зв'язку на основі опуклих оболонок зон покриття.

8. Створення програми для аналізу та візуалізації астрономічних даних з виділенням опуклих оболонок зоряних скупчень.

9. Реалізація модуля стиснення векторної графіки шляхом апроксимації складних контурів опуклими оболонками.

10. Розробка системи моніторингу та аналізу руху об'єктів з використанням опуклих оболонок траєкторій для виявлення аномалій.

11.1.2. Алгоритм Джарвіса

Теоретичні відомості

Алгоритм Джарвіса, також відомий як алгоритм загортання подарунка (gift wrapping algorithm), був розроблений Р. А. Джарвісом у 1973 році. Це інтуїтивно зрозумілий метод побудови опуклої оболонки, який імітує процес загортання подарунка в папір. Уявіть, що у вас є набір цвяхів, забитих у дошку, і ви обгортаєте їх мотузкою, натягуючи її якомога туже. Форма, яку прийме мотузка, і буде опуклою оболонкою.

Основна ідея алгоритму полягає в послідовному знаходженні вершин опуклої оболонки, рухаючись по її периметру. Починаючи з точки, яка гарантовано належить опуклій оболонці (наприклад, найлівішої точки), алгоритм на кожному кроці знаходить наступну вершину оболонки, вибираючи точку, яка утворює найбільший кут з попереднім ребром.

Ключовим елементом алгоритму є визначення того, яка точка буде наступною на опуклій оболонці. Для цього використовується поняття орієнтації трьох точок та полярного кута.

Орієнтація трьох точок

Для трьох точок $P(x_1, y_1)$, $Q(x_2, y_2)$ та $R(x_3, y_3)$ орієнтацію визначаємо через знак виразу:

$$\text{orientation} = (y_2 - y_1)(x_3 - x_2) - (x_2 - x_1)(y_3 - y_2)$$

Цей вираз є подвоєною знаковою площею трикутника PQR. Інтерпретація:
- orientation > 0: точки утворюють лівий поворот (проти годинникової

стрілки) - orientation < 0: точки утворюють правий поворот (за годинниковою стрілкою)

- orientation = 0: точки колінеарні

Пошук найправішої точки

На кожному кроці алгоритму, маючи поточну точку P на опуклій оболонці, потрібно знайти наступну точку Q таку, що всі інші точки лежать зліва від променя PQ (або на ньому). Це еквівалентно знаходженню точки Q, для якої всі трійки (P, Q, R) мають невід'ємну орієнтацію для всіх інших точок R.

Детальний опис алгоритму такий:

```
Алгоритм JarvisMarch(points):
    Вхід: points - масив з n точок на площині
    Вихід: hull - масив точок опуклої оболонки в порядку обходу

    // Крок 1: Знайти стартову точку (найлівішу)
    start = FindLeftmostPoint(points)

    // Крок 2: Ініціалізація
    hull = []
    current = start

    // Крок 3: Основний цикл - обхід по периметру
    повторювати:
        hull.додати(current)

        // Знайти наступну точку на оболонці
        next = points[0]
        для кожної точки p в points:
            якщо p == current:
                продовжити

            // Перевірити, чи p правіше за next відносно current
            якщо next == current або IsMoreCounterClockwise(current,
p, next):
                next = p

        current = next

    поки current != start  // Повернулися до початку

    повернути hull

Функція FindLeftmostPoint(points):
```

```
    Вхід: points - масив точок
    Вихід: найлівіша точка (з найменшою x-координатою)

    leftmost = points[0]

    для кожної точки p в points:
        якщо p.x < leftmost.x або (p.x == leftmost.x та p.y < leftmos
t.y):
            leftmost = p

    повернути leftmost

Функція IsMoreCounterClockwise(origin, p1, p2):
    Вхід: origin - початкова точка, p1, p2 - точки для порівняння
    Вихід: true, якщо p1 лежить проти годинникової стрілки від p2 від
носно origin

    // Обчислити орієнтацію
    orientation = Orientation(origin, p2, p1)

    якщо orientation > 0:
        // p1 зліва від origin->p2
        повернути true
    інакше якщо orientation < 0:
        // p1 справа від origin->p2
        повернути false
    інакше:
        // Точки колінеарні, вибираємо дальшу
        dist1 = DistanceSquared(origin, p1)
        dist2 = DistanceSquared(origin, p2)
        повернути dist1 > dist2

Функція Orientation(p, q, r):
    Вхід: три точки p, q, r
    Вихід: знакова подвоєна площа трикутника pqr

    повернути (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y)

Функція DistanceSquared(p1, p2):
    Вхід: дві точки p1, p2
    Вихід: квадрат евклідової відстані

    dx = p2.x - p1.x
    dy = p2.y - p1.y
    повернути dx * dx + dy * dy
```

Алгоритм потребує спеціальної уваги до кількох особливих випадків:

Колінеарні точки на межі оболонки

Коли кілька точок лежать на одній прямій з поточною точкою, потрібно вибрати найдальшу з них (якщо рухаємося від попередньої точки) або найближчу (на останньому кроці при поверненні до початкової точки).

```
Функція HandleCollinearPoints(origin, candidates, isLastStep):
    Вхід:
        origin - поточна точка
        candidates - масив колінеарних точок
        isLastStep - чи це останній крок (повернення до початку)
    Вихід: правильно вибрана точка

    якщо isLastStep:
        // На останньому кроці вибираємо найближчу точку
        closest = candidates[0]
        minDist = DistanceSquared(origin, closest)

        для кожної точки p в candidates:
            dist = DistanceSquared(origin, p)
            якщо dist < minDist:
                minDist = dist
                closest = p

        повернути closest
    інакше:
        // В інших випадках вибираємо найдальшу точку
        farthest = candidates[0]
        maxDist = DistanceSquared(origin, farthest)

        для кожної точки p в candidates:
            dist = DistanceSquared(origin, p)
            якщо dist > maxDist:
                maxDist = dist
                farthest = p

        повернути farthest
```

Вироджені випадки

- Менше 3 точок: опукла оболонка - це самі точки

- Всі точки колінеарні: опукла оболонка - це відрізок між крайніми точками

- Дублікати точок: потрібно або видалити дублікати, або коректно їх обробляти

Розглянемо роботу алгоритму на наборі з 8 точок:

```
Точки: A(1,1), B(3,4), C(5,2), D(6,5), E(4,6), F(2,5), G(0,3), H(3,0)
```

Крок 1: Знаходимо стартову точку

```
Перевіряємо x-координати:
G(0,3) має найменшу x-координату: 0
Стартова точка: G(0,3)
```

Візуалізація початкового стану:

```
  6 |    E
  5 |  F   D
  4 |  B
  3 |G
  2 |    C
  1 |A
  0 |  H
    +----------
    0123456
```

Крок 2: Перша ітерація (з точки G)

Шукаємо точку, така що всі інші точки лежать зліва від променя з G:

```
Перевіряємо кожну точку як кандидата:
- G→A: перевіряємо орієнтацію (G,A,B), (G,A,C), тощо
- G→B: перевіряємо орієнтацію (G,B,A), (G,B,C), тощо
...

Для G→A:
Orientation(G,A,B) = (1-3)*(4-3) - (1-0)*(3-1) = -2*1 - 1*2 = -4 < 0
Точка B справа від GA, тому A не підходить

Для G→H:
Перевіряємо всі точки - всі дають невід'ємну орієнтацію
H - наступна точка на оболонці
```

Крок 3: Друга ітерація (з точки H)

```
Поточна точка: H(3,0)
Шукаємо наступну точку...

Після перевірки всіх кандидатів: C(5,2) - наступна точка
```

Продовження ітерацій:

```
Ітерація 3: C → D
Ітерація 4: D → E
Ітерація 5: E → F
Ітерація 6: F → G (повернення до початку)
```

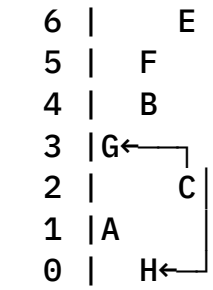
Візуалізація кінцевого результату:

```
  6 |    E*
  5 |  F*  D*
  4 |  B
  3 |G*
  2 |    C*
  1 |A
  0 |  H*
    +----------
    0123456
```

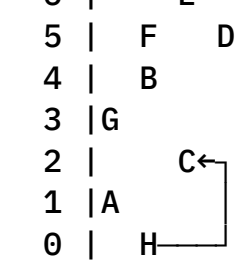
Опукла оболонка: G → H → C → D → E → F → G
Точки всередині: A, B

Покрокова візуалізація процесу “загортання”:

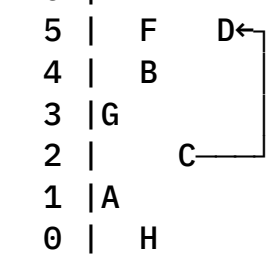
Крок 1: G обрана як старт



Крок 2: G→H



Крок 3: H→C



І так далі...

Аналіз складності

Часова складність: - Пошук стартової точки: O(n) - Основний цикл: O(h) ітерацій, де h - кількість точок на опуклій оболонці - На кожній ітерації: O(n) для пошуку наступної точки - Загальна складність: O(nh)

Це робить алгоритм Джарвіса ефективним, коли h << n (кількість точок на оболонці набагато менша за загальну кількість точок).

Просторова складність: O(h) для зберігання результату. Алгоритм працює “на місці”, не потребуючи додаткових структур даних.

Оптимізації алгоритму

1. Використання полярних координат

Замість перевірки орієнтації для кожної пари точок, можна обчислити полярні кути:

```
Функція JarvisMarchOptimized(points):
    start = FindLeftmostPoint(points)
    hull = []
    current = start
    previous = Точка(current.x, current.y - 1)  // Умовна точка знизу

    повторювати:
        hull.додати(current)
        next = null
        maxAngle = -∞

        для кожної точки p в points:
            якщо p == current:
                продовжити

            angle = PolarAngle(current, previous, p)
            якщо angle > maxAngle:
                maxAngle = angle
                next = p

        previous = current
        current = next

    поки current != start

    повернути hull

Функція PolarAngle(origin, from, to):
    // Обчислює кут між векторами origin→from та origin→to
    v1 = (from.x - origin.x, from.y - origin.y)
```

```
    v2 = (to.x - origin.x, to.y - origin.y)

    // Використовуємо atan2 для правильної обробки всіх квадрантів
    angle1 = atan2(v1.y, v1.x)
    angle2 = atan2(v2.y, v2.x)

    diff = angle2 - angle1
    якщо diff < 0:
        diff += 2π

    повернути diff
```

Область застосування

Алгоритм Джарвіса знаходить застосування в багатьох галузях завдяки своїй простоті та ефективності для певних типів даних:

- Комп'ютерна графіка та моделювання: Визначення видимих меж об'єктів, побудова обмежувальних контурів для прискорення рендерингу, створення спрощених моделей для фізичних симуляцій

- Обробка зображень: Виділення контурів об'єктів на зображеннях, сегментація зображень, розпізнавання форм та жестів

- Геоінформаційні системи: Визначення меж географічних регіонів, створення буферних зон навколо об'єктів, аналіз просторового розподілу

- Робототехніка та навігація: Планування безпечних маршрутів для роботів, визначення зон досяжності маніпуляторів, уникнення перешкод

- Аналіз даних та статистика: Виявлення викидів у наборах даних, визначення меж кластерів, візуалізація багатовимірних даних після зниження розмірності

- Ігрова індустрія: Обчислення зон видимості в стратегічних іграх, визначення меж територій, спрощення колізій

- Метеорологія та екологія: Визначення зон поширення забруднень, моделювання погодних фронтів, аналіз міграції тварин

Приклади застосування

Приклад 1: Визначення зони покриття мобільної мережі

Телекомунікаційна компанія розміщує базові станції та хоче визначити загальну зону покриття:

```
Алгоритм CoverageZoneAnalysis(stations, coverage_radius):
    // Генеруємо точки на межі покриття кожної станції
    boundary_points = []
```

```
    для кожної станції s в stations:
        // Генеруємо точки по колу навколо станції
        для angle від 0 до 2π з кроком π/18:  // 36 точок
            x = s.x + coverage_radius * cos(angle)
            y = s.y + coverage_radius * sin(angle)
            boundary_points.додати(Точка(x, y))

    // Знаходимо опуклу оболонку всіх точок покриття
    coverage_hull = JarvisMarch(boundary_points)

    // Аналіз результатів
    total_area = CalculatePolygonArea(coverage_hull)
    efficiency = total_area / (π * coverage_radius² * кількість_станц
ій)

    повернути {
        hull: coverage_hull,
        area: total_area,
        efficiency: efficiency
    }
```

Приклад 2: Система виявлення аномалій у фінансових транзакціях

Банк аналізує транзакції клієнтів у двовимірному просторі (сума, частота):

```
Алгоритм AnomalyDetection(transactions):
    // Нормалізуємо дані
    normalized = NormalizeData(transactions)

    // Знаходимо опуклу оболонку "нормальних" транзакцій
    // (виключаємо 5% найвіддаленіших точок)
    sorted_by_distance = SortByDistanceFromCenter(normalized)
    normal_count = floor(0.95 * довжина(sorted_by_distance))
    normal_transactions = sorted_by_distance[0:normal_count]

    normal_hull = JarvisMarch(normal_transactions)

    // Виявляємо аномалії
    anomalies = []
    для кожної транзакції t в transactions:
        якщо не IsPointInPolygon(t, normal_hull):
            anomalies.додати(t)

    повернути anomalies

Функція IsPointInPolygon(point, polygon):
```

```
    // Алгоритм променевого сканування
    intersections = 0
    n = довжина(polygon)

    для і від 0 до n-1:
        p1 = polygon[i]
        p2 = polygon[(i+1) % n]

        якщо RayIntersectsSegment(point, p1, p2):
            intersections += 1

    повернути intersections % 2 == 1
```

Приклад 3: Оптимізація розміщення складів для логістичної компанії

Компанія має набір точок доставки та хоче визначити оптимальні зони обслуговування:

```
Алгоритм WarehouseZoneOptimization(delivery_points):
    // Знаходимо опуклу оболонку всіх точок доставки
    hull = JarvisMarch(delivery_points)

    // Визначаємо характеристики зони
    perimeter = CalculatePerimeter(hull)
    area = CalculateArea(hull)
    center = CalculateCentroid(hull)

    // Оцінка ефективності покриття
    actual_points_area = π * середній_радіус² * кількість_точок
    coverage_efficiency = actual_points_area / area

    // Рекомендація щодо кількості складів
    якщо area > LARGE_AREA_THRESHOLD:
        // Розбити на підзони
        k = ceil(area / OPTIMAL_ZONE_SIZE)
        subzones = KMeansClustering(delivery_points, k)

        warehouse_locations = []
        для кожної зони в subzones:
            zone_hull = JarvisMarch(зона.points)
            zone_center = CalculateCentroid(zone_hull)
            warehouse_locations.додати(zone_center)
    інакше:
        warehouse_locations = [center]

    повернути {
        hull: hull,
```

```
        warehouse_locations: warehouse_locations,
        efficiency: coverage_efficiency
    }
```

Переваги та недоліки

Переваги:

- Простота реалізації: Алгоритм має інтуїтивну логіку “загортання подарунка”, легко зрозуміти та реалізувати

- Висока ефективність для малих опуклих оболонок: При $h << n$ алгоритм працює майже лінійно - $O(nh) \approx O(n)$

- Робота “на місці”: Не потребує додаткових структур даних крім масиву для результату

- Адаптивність: Час роботи залежить від фактичної кількості точок на оболонці, а не від загальної кількості

- Стійкість: Не чутливий до розподілу точок, працює однаково для будь-якої конфігурації

- Можливість ранньої зупинки: Можна зупинити алгоритм після знаходження k точок оболонки

Недоліки:

- Квадратична складність у гіршому випадку: Коли всі точки лежать на опуклій оболонці, $O(nh) = O(n^2)$

- Повільніший за оптимальні алгоритми: Для загального випадку програє алгоритмам зі складністю $O(n \log n)$

- Чутливість до точності обчислень: Порівняння кутів може давати помилки через обмежену точність

- Складність паралелізації: Послідовна природа алгоритму ускладнює розпаралелювання

- Неефективність для 3D: Узагальнення на тривимірний простір має складність $O(n^2)$

- Відсутність інформації про внутрішню структуру: Не дає інформації про точки всередині оболонки

Контрольні запитання

1. Поясніть основну ідею алгоритму Джарвіса та чому він називається “алгоритмом загортання подарунка”.

2. Яка часова складність алгоритму Джарвіса та від чого вона залежить?

3. Як визначається наступна точка опуклої оболонки на кожному кроці алгоритму?

4. Чому алгоритм починає з найлівішої (або найнижчої) точки?

5. Як обробляються колінеарні точки в алгоритмі Джарвіса?

6. Порівняйте алгоритм Джарвіса з алгоритмом сканування Грема за ефективністю.

7. В яких випадках алгоритм Джарвіса є оптимальним вибором?

8. Як можна модифікувати алгоритм для пошуку опуклої оболонки в 3D просторі?

9. Які числові проблеми можуть виникнути при реалізації алгоритму?

10. Як адаптувати алгоритм для роботи з цілочисельною арифметикою?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Джарвіса для побудови опуклої оболонки

2. Мета

Вивчити принцип роботи алгоритму загортання подарунка, освоїти техніки пошуку екстремальних точок, навчитися ефективно реалізовувати геометричні алгоритми з адаптивною складністю

3. Завдання

- Реалізувати базову версію алгоритму Джарвіса

- Створити функції для роботи з геометричними примітивами

- Додати обробку особливих випадків

- Реалізувати оптимізовану версію без тригонометричних функцій

- Створити візуалізацію процесу побудови

- Провести порівняльний аналіз з іншими алгоритмами

- Реалізувати модифікацію для часткової побудови оболонки

4. Опис

Лабораторна робота передбачає реалізацію класичного алгоритму Джарвіса з поетапною візуалізацією процесу “загортання”. Особлива увага приділяється обробці граничних випадків та оптимізації для різних типів розподілу точок.

5. Порядок виконання

- Створити структури даних для точок та векторів
- Реалізувати функцію обчислення орієнтації трьох точок
- Написати функцію пошуку крайніх точок (найлівіша, найправіша тощо)
- Реалізувати основний цикл алгоритму Джарвіса
- Додати обробку колінеарних точок
- Створити модуль візуалізації з анімацією “мотузки”
- Реалізувати генератори тестових даних різних типів
- Додати можливість покрокового виконання
- Провести вимірювання продуктивності
- Порівняти з іншими алгоритмами побудови опуклої оболонки

6. Результат

Інтерактивна програма, що демонструє роботу алгоритму Джарвіса з можливістю візуалізації кожного кроку, порівняння з іншими методами та аналізу ефективності для різних типів даних.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
- Модуль геометричних примітивів (точка, вектор, відрізок)
- Функція визначення орієнтації трьох точок
- Функція пошуку крайніх точок
- Основний алгоритм Джарвіса
- Оптимізована версія без тригонометрії
- Модуль обробки колінеарних точок
- Візуалізатор з анімацією
- Генератори тестових даних
- Модуль статистики та аналізу
- Продемонструвати роботу додатку:
- Покрокове виконання алгоритму з візуалізацією
- Анімація процесу “загортання подарунка”
- Робота з різними типами розподілу точок
- Обробка вироджених випадків

- Порівняння часу виконання з алгоритмом Грема
- Демонстрація випадків, коли алгоритм Джарвіса ефективніший
- Часткова побудова оболонки (перші k точок)

Бути готовим відповісти на питання:

- Чому алгоритм має адаптивну складність?
- Як уникнути числових помилок при порівнянні кутів?
- Коли алгоритм Джарвіса кращий за інші методи?
- Як реалізувати 3D версію алгоритму?
- Які оптимізації можливі для спеціальних випадків?
- Як використати алгоритм для динамічних даних?

Тести

1. Яка часова складність алгоритму Джарвіса?

1) O(n log n)

2) O(nh)

3) $O(n^2)$

4) $O(h^2)$

2. Яку точку обирають як стартову в алгоритмі Джарвіса?

1) Випадкову точку

2) Точку з найбільшою y-координатою

3) Крайню точку (наприклад, найлівішу)

4) Центр мас всіх точок

3. Скільки точок перевіряється на кожній ітерації алгоритму?

1) Лише сусідні точки

2) Всі точки

3) Лише точки на поточній оболонці

4) Половина точок

4. Що означає h у складності O(nh)?

1) Висота дерева

2) Кількість точок на опуклій оболонці

3) Кількість ітерацій

4) Хеш-значення

5. Як визначається наступна точка оболонки?

1) Найближча до поточної

2) З максимальною y-координатою

3) Така, що всі інші точки зліва від променя

4) Випадкова з решти точок

Відповіді:

1. 2)

2. 3)

3. 2)

4. 2)

5. 3)

Тести підвищеної складності

1. У якому випадку алгоритм Джарвіса матиме складність $O(n^2)$?

1) Коли точки розподілені випадково

2) Коли всі точки лежать на опуклій оболонці

3) Коли точки утворюють правильний многокутник

4) Коли є багато дублікатів

2. Як правильно обробити випадок трьох колінеарних точок A, B, C на одному промені з поточної точки?

1) Вибрати середню точку B

2) Вибрати найближчу точку

3) Вибрати найдальшу точку (звичайно) або найближчу (на останньому кроці)

4) Включити всі три точки в оболонку

3. Яка структура даних найкраще підходить для оптимізації пошуку наступної точки?

1) KD-дерево

2) Хеш-таблиця

3) Не потрібна додаткова структура

4) R-дерево

4. Скільки разів кожна точка розглядається як кандидат протягом всього алгоритму?

1) Рівно один раз

2) Максимум h разів

3) Рівно h разів

4) Від 1 до n разів

5. Як можна модифікувати алгоритм для пошуку k найлівіших точок оболонки?

1) Зупинити алгоритм після k ітерацій

2) Це неможливо з алгоритмом Джарвіса

3) Спочатку побудувати всю оболонку, потім взяти k точок

4) Використати інший алгоритм

6. Що станеться, якщо всі точки однакові?

1) Алгоритм зациклиться

2) Повернеться порожня оболонка

3) Повернеться одна точка

4) Виникне помилка ділення на нуль

7. Як уникнути використання тригонометричних функцій?

1) Використати лише цілочисельну арифметику

2) Порівнювати орієнтації замість кутів

3) Наближено обчислювати кути

4) Це неможливо

8. Для якого розподілу точок алгоритм Джарвіса найефективніший?

1) Рівномірний розподіл у квадраті

2) Точки на колі

3) Нормальний розподіл з малою дисперсією

4) Точки на решітці

9. Яка модифікація потрібна для побудови опуклої оболонки “з дірками” (без внутрішніх точок на межі)?

1) Змінити нестрогі нерівності на строгі при перевірці орієнтації

2) Видалити колінеарні точки після побудови

3) Використати інший початковий напрямок

4) Додати перевірку на колінеарність

10. Як адаптувати алгоритм для потокових даних?

1) Неможливо, потрібен інший алгоритм

2) Перезапускати для кожної нової точки

3) Перевіряти лише, чи нова точка розширює оболонку

4) Зберігати всі проміжні результати

Відповіді:

1. 2) Коли всі n точок лежать на опуклій оболонці, h = n, тому $O(nh) = O(n^2)$

2. 3) Звичайно вибираємо найдальшу для максимізації прогресу, але на останньому кроці - найближчу

3. 3) Алгоритм Джарвіса не потребує попередньої обробки або додаткових структур

4. 3) Кожна точка перевіряється на кожній з h ітерацій алгоритму

5. 1) Можна зупинити алгоритм після знаходження k точок, оскільки він будує оболонку послідовно

6. 3) Алгоритм коректно обробить цей випадок і поверне одну точку

7. 2) Використання орієнтації (векторного добутку) дозволяє уникнути тригонометрії

8. 3) Коли більшість точок зосереджені в центрі, h буде малим, що дає O(n)

9. 1) Строгі нерівності виключать колінеарні точки з внутрішньої частини ребер

10. 3) Можна ефективно перевірити, чи розширює нова точка існуючу оболонку за O(h)

Завдання для самостійної роботи

1. Реалізуйте гібридний алгоритм, який автоматично перемикається між алгоритмами Джарвіса та Грема залежно від оціночної кількості точок на оболонці.

2. Розробіть паралельну версію алгоритму Джарвіса, розділивши площину на сектори та об’єднуючи часткові результати.

3. Створіть модифікацію алгоритму для побудови k-опуклої оболонки (оболонки, що містить принаймні k точок).

4. Реалізуйте алгоритм Джарвіса для опуклої оболонки на сфері, адаптувавши поняття “лівого повороту” для сферичної геометрії.

5. Розробіть інкрементальну версію, яка ефективно оновлює опуклу оболонку при додаванні нових точок.

6. Створіть алгоритм для знаходження всіх опуклих шарів набору точок, використовуючи алгоритм Джарвіса.

7. Реалізуйте версію з наближеними обчисленнями для роботи з великими наборами даних у реальному часі.

8. Розробіть модифікацію для побудови ортогональної опуклої оболонки (з ребрами паралельними осям).

9. Створіть адаптивну версію, яка використовує евристики для прискорення пошуку наступної точки.

10. Реалізуйте алгоритм для динамічної опуклої оболонки з підтримкою видалення точок.

Теми лабораторних робіт

1. Розробка інтерактивного візуалізатора алгоритмів побудови опуклої оболонки з порівнянням Джарвіса, Грема та QuickHull.

2. Створення системи аналізу траєкторій рухомих об’єктів з використанням опуклих оболонок для виявлення аномалій.

3. Реалізація геометричного процесора для обробки географічних даних з підтримкою різних алгоритмів опуклої оболонки.

4. Розробка гри “Захоплення території”, де гравці будують опуклі оболонки навколо точок на карті.

5. Створення системи оптимізації логістичних маршрутів на основі аналізу опуклих оболонок точок доставки.

6. Реалізація модуля комп’ютерного зору для розпізнавання форм об’єктів через їх опуклі оболонки.

7. Розробка інструменту для аналізу кластерів у багатовимірних даних з проекцією та побудовою опуклих оболонок.

8. Створення симулятора фізичних процесів, який використовує опуклі оболонки для виявлення колізій.

9. Реалізація системи планування покриття для бездротових мереж на основі опуклих оболонок зон обслуговування.

10. Розробка редактора векторної графіки з автоматичним спрощенням складних фігур до опуклих оболонок.

11.1.3. Алгоритм QuickHull

Теоретичні відомості

Алгоритм QuickHull — це метод побудови опуклої оболонки, який використовує стратегію “розділяй і володарюй”. Розроблений у 1970-х роках, цей алгоритм отримав свою назву за аналогією з алгоритмом швидкого сортування (QuickSort), оскільки використовує подібний принцип рекурсивного розділення задачі на менші підзадачі.

Основна ідея QuickHull полягає в тому, щоб знайти крайні точки опуклої оболонки, побудувати відрізок між ними, а потім рекурсивно обробити точки, що лежать по різні боки від цього відрізка. На кожному кроці алгоритм знаходить точку, найвіддаленішу від поточного ребра, і будує трикутник. Точки всередині цього трикутника відкидаються, оскільки вони гарантовано не належать опуклій оболонці.

Геометричні основи

Відстань від точки до прямої

Ключовим елементом алгоритму є обчислення відстані від точки до прямої. Для точки $P(x_0, y_0)$ та прямої, що проходить через точки $A(x_1, y_1)$ та $B(x_2, y_2)$, знакова відстань обчислюється за формулою:

$$d = ((y_2 - y_1) * x_0 - (x_2 - x_1) * y_0 + x_2 * y_1 - y_2 * x_1) / \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

Однак для цілей алгоритму достатньо використовувати чисельник цієї формули (без нормалізації), оскільки нас цікавить лише порівняння відстаней:

$$d\_signed = (y_2 - y_1) * x_0 - (x_2 - x_1) * y_0 + x_2 * y_1 - y_2 * x_1$$

Знак цієї величини вказує, з якого боку від прямої AB знаходиться точка P: - d_signed > 0: точка зліва від напрямленого відрізка AB - d_signed < 0: точка справа від напрямленого відрізка AB - d_signed = 0: точка лежить на прямій AB

Площа трикутника

Абсолютне значення d_signed дорівнює подвоєній площі трикутника ABP. Це властивість використовується для визначення, чи лежить точка всередині трикутника.

Наведемо опис алгоритму:

```
Алгоритм QuickHull(points):
    Вхід: points - масив точок на площині
    Вихід: convex_hull - список точок опуклої оболонки

    якщо кількість точок < 3:
        повернути points

    // Крок 1: Знайти крайні точки
    leftmost = FindLeftmostPoint(points)
    rightmost = FindRightmostPoint(points)

    // Крок 2: Розділити точки на дві групи
    upper_points = []
    lower_points = []

    для кожної точки p в points:
        якщо p == leftmost або p == rightmost:
            продовжити

        side = GetSide(leftmost, rightmost, p)
        якщо side > 0:
            upper_points.додати(p)
        інакше якщо side < 0:
            lower_points.додати(p)

    // Крок 3: Побудувати верхню та нижню частини оболонки
    convex_hull = []
    convex_hull.додати(leftmost)

    // Верхня частина
    upper_hull = FindHull(leftmost, rightmost, upper_points)
    convex_hull.додати_всі(upper_hull)

    convex_hull.додати(rightmost)

    // Нижня частина
    lower_hull = FindHull(rightmost, leftmost, lower_points)
    convex_hull.додати_всі(lower_hull)
```

```
    повернути convex_hull

Функція FindHull(p1, p2, points):
    Вхід:
        p1, p2 - кінці поточного ребра
        points - точки з одного боку від ребра p1-p2
    Вихід: список точок опуклої оболонки між p1 та p2

    якщо points порожній:
        повернути []

    // Знайти найвіддаленішу точку від відрізка p1-p2
    farthest = null
    max_distance = 0

    для кожної точки p в points:
        distance = abs(GetSignedDistance(p1, p2, p))
        якщо distance > max_distance:
            max_distance = distance
            farthest = p

    // Видалити точки всередині трикутника p1-farthest-p2
    left_points = []
    right_points = []

    для кожної точки p в points:
        якщо p == farthest:
            продовжити

        якщо IsPointInTriangle(p1, farthest, p2, p):
            продовжити  // Точка всередині трикутника - відкидаємо

        // Визначити, з якого боку точка
        якщо GetSide(p1, farthest, p) > 0:
            left_points.додати(p)
        інакше якщо GetSide(farthest, p2, p) > 0:
            right_points.додати(p)

    // Рекурсивно обробити обидві сторони
    hull = []
    left_hull = FindHull(p1, farthest, left_points)
    hull.додати_всі(left_hull)

    hull.додати(farthest)
```

```
    right_hull = FindHull(farthest, p2, right_points)
    hull.додати_всі(right_hull)

    повернути hull
```

Допоміжні функції:

```
Функція FindLeftmostPoint(points):
    leftmost = points[0]
    для кожної точки p в points:
        якщо p.x < leftmost.x або (p.x == leftmost.x та p.y < leftmos
t.y):
            leftmost = p
    повернути leftmost

Функція FindRightmostPoint(points):
    rightmost = points[0]
    для кожної точки p в points:
        якщо p.x > rightmost.x або (p.x == rightmost.x та p.y > right
most.y):
            rightmost = p
    повернути rightmost

Функція GetSide(p1, p2, p):
    // Повертає знакову відстань (подвоєну площу трикутника)
    повернути (p2.y - p1.y) * p.x - (p2.x - p1.x) * p.y + p2.x * p1.y
- p2.y * p1.x

Функція GetSignedDistance(p1, p2, p):
    повернути GetSide(p1, p2, p)

Функція IsPointInTriangle(a, b, c, p):
    // Перевіряє, чи точка p всередині трикутника abc
    // Використовуємо барицентричні координати

    v0x = c.x - a.x
    v0y = c.y - a.y
    v1x = b.x - a.x
    v1y = b.y - a.y
    v2x = p.x - a.x
    v2y = p.y - a.y

    dot00 = v0x * v0x + v0y * v0y
    dot01 = v0x * v1x + v0y * v1y
    dot02 = v0x * v2x + v0y * v2y
```

```
    dot11 = v1x * v1x + v1y * v1y
    dot12 = v1x * v2x + v1y * v2y

    invDenom = 1 / (dot00 * dot11 - dot01 * dot01)
    u = (dot11 * dot02 - dot01 * dot12) * invDenom
    v = (dot00 * dot12 - dot01 * dot02) * invDenom

    повернути (u >= 0) та (v >= 0) та (u + v <= 1)
```

Оптимізації алгоритму

1. Попередня фільтрація точок

Перед початком основного алгоритму можна відфільтрувати точки, які точно не належать опуклій оболонці:

```
Функція PreprocessPoints(points):
    // Знайти обмежувальний прямокутник
    min_x = min_y = нескінченність
    max_x = max_y = -нескінченність

    для кожної точки p в points:
        min_x = min(min_x, p.x)
        max_x = max(max_x, p.x)
        min_y = min(min_y, p.y)
        max_y = max(max_y, p.y)

    // Знайти кутові точки
    corners = []
    для кожної точки p в points:
        якщо (p.x == min_x або p.x == max_x) та (p.y == min_y або p.y
== max_y):
            corners.додати(p)

    // Відфільтрувати точки всередині опуклої оболонки кутових точок
    filtered = []
    для кожної точки p в points:
        якщо не IsInsidePolygon(p, corners):
            filtered.додати(p)

    повернути filtered
```

Розглянемо роботу алгоритму на наборі з 10 точок:

```
Точки: A(1,2), B(3,5), C(5,3), D(7,4), E(6,7),
       F(4,8), G(2,6), H(0,4), I(4,1), J(8,5)
```

Крок 1: Знаходимо крайні точки

```
Візуалізація:
  8 |    F
  7 |      E
  6 |  G
  5 |  B     J
  4 |H     D
  3 |    C
  2 |A
  1 |    I
  0 +----------
    012345678

Leftmost: H(0,4)
Rightmost: J(8,5)
```

Крок 2: Розділяємо точки відносно лінії H-J

```
Лінія H-J розділяє площину:
- Верхні точки (зліва від H→J): B, F, G, E
- Нижні точки (справа від H→J): A, C, D, I
```

Крок 3: Обробка верхньої частини

Рекурсивний виклик FindHull(H, J, {B, F, G, E}):

```
Знаходимо найвіддаленішу точку від H-J:
- Відстань B: d = |GetSide(H, J, B)| = 24
- Відстань F: d = |GetSide(H, J, F)| = 32
- Відстань G: d = |GetSide(H, J, G)| = 16
- Відстань E: d = |GetSide(H, J, E)| = 24

Найвіддаленіша: F(4,8)
```

Будуємо трикутник H-F-J:

```
  8 |    F*
  7 |   / \E
  6 | /G   \
  5 |/  B   \J
  4 |H-------
```

Точки B, G, E перевіряємо на належність трикутнику: - B: всередині трикутника → відкидаємо - G: зовні, зліва від H-F - E: зовні, справа від F-J

Рекурсивні виклики: - FindHull(H, F, {G}) → повертає [G] - FindHull(F, J, {E}) → повертає [E]

Верхня частина: [G, F, E]

Крок 4: Обробка нижньої частини

Аналогічно обробляємо нижні точки для FindHull(J, H, {A, C, D, I}).

Фінальний результат:

```
Опукла оболонка: H → G → F → E → J → D → I → A → H

Візуалізація:
  8 |    F*
  7 |   /|\E*
  6 | G* | /
  5 |  \ |/ J*
  4 |H*-\|/-D*
  3 |    C
  2 |A*--/
  1 |    I*
```

Аналіз складності

Часова складність: - Найкращий випадок: O(n log n) — коли точки розподілені рівномірно - Середній випадок: O(n log n) - Найгірший випадок: $O(n^2)$ — коли всі точки лежать на опуклій оболонці

Глибина рекурсії залежить від розподілу точок. У найкращому випадку на кожному рівні рекурсії кількість точок зменшується вдвічі, що дає log n рівнів.

Просторова складність: O(n) для зберігання проміжних результатів та стеку рекурсії.

Порівняння з іншими алгоритмами

QuickHull поєднує переваги різних підходів:

- Як і алгоритм Джарвіса, може бути ефективним для малих опуклих оболонок

- Як і алгоритм Грема, має хорошу середню продуктивність

- Використовує стратегію “розділяй і володарюй”, що робить його інтуїтивно зрозумілим

Область застосування

Алгоритм QuickHull широко використовується завдяки своїй ефективності та простоті реалізації:

- 3D моделювання та комп’ютерна графіка: QuickHull легко узагальнюється на тривимірний простір, що робить його популярним для побудови опуклих оболонок 3D об’єктів

- Обробка геопросторових даних: Визначення меж територій, створення буферних зон, аналіз кластерів географічних об’єктів

- Фізичні симуляції: Виявлення колізій між складними об’єктами через їх опуклі оболонки, моделювання твердих тіл

- Машинне навчання: Візуалізація та аналіз кластерів даних, визначення меж класів у просторі ознак

- Робототехніка: Планування траєкторій руху, визначення досяжних зон маніпуляторів

- Комп’ютерний зір: Розпізнавання форм об’єктів, виділення контурів, аналіз сцен

- Оптимізаційні задачі: Лінійне програмування, пошук екстремальних точок у багатовимірних просторах

Приклади застосування

Приклад 1: Виявлення колізій у грі

У комп’ютерній грі потрібно перевіряти зіткнення складних об’єктів:

```
Алгоритм CollisionDetection(object1_points, object2_points):
    // Побудова опуклих оболонок
    hull1 = QuickHull(object1_points)
    hull2 = QuickHull(object2_points)

    // Швидка перевірка через обмежувальні прямокутники
    bbox1 = GetBoundingBox(hull1)
    bbox2 = GetBoundingBox(hull2)

    якщо не BoundingBoxesIntersect(bbox1, bbox2):
        повернути false  // Немає колізії

    // Детальна перевірка SAT (Separating Axis Theorem)
    для кожного ребра edge в hull1:
        axis = GetNormal(edge)
        projection1 = ProjectPolygon(hull1, axis)
        projection2 = ProjectPolygon(hull2, axis)

        якщо не ProjectionsOverlap(projection1, projection2):
            повернути false  // Знайдена розділяюча вісь

    // Аналогічно для ребер hull2
    для кожного ребра edge в hull2:
        // ... аналогічна перевірка

    повернути true  // Колізія виявлена
```

```
Функція ProjectPolygon(polygon, axis):
    min_proj = max_proj = DotProduct(polygon[0], axis)

    для і від 1 до довжина(polygon)-1:
        proj = DotProduct(polygon[i], axis)
        min_proj = min(min_proj, proj)
        max_proj = max(max_proj, proj)

    повернути (min_proj, max_proj)
```

Приклад 2: Аналіз форми хмар точок у лідарних даних

Система автономного водіння аналізує дані з лідара:

```
Алгоритм LidarObjectClassification(point_cloud):
    // Сегментація хмари точок на об'єкти
    objects = SegmentPointCloud(point_cloud)

    classified_objects = []

    для кожного об'єкта obj в objects:
        // Проекція на площину XY
        xy_points = ProjectToXY(obj.points)

        // Побудова опуклої оболонки
        hull = QuickHull(xy_points)

        // Обчислення характеристик форми
        area = CalculateArea(hull)
        perimeter = CalculatePerimeter(hull)
        compactness = 4 * π * area / (perimeter²)

        // Обчислення орієнтованого обмежувального прямокутника
        obb = MinimumBoundingRectangle(hull)
        aspect_ratio = obb.length / obb.width

        // Класифікація на основі характеристик
        якщо obj.height > 1.5 та aspect_ratio > 2.5:
            type = "Автомобіль"
        інакше якщо obj.height > 1.2 та compactness > 0.7:
            type = "Людина"
        інакше якщо obj.height < 0.5 та area > 10:
            type = "Дорожня розмітка"
        інакше:
            type = "Невідомий"

        classified_objects.додати({
```

```
            object: obj,
            type: type,
            hull: hull,
            features: {area, perimeter, compactness, aspect_ratio}
        })

    повернути classified_objects
```

Приклад 3: Оптимізація розміщення антен

Телекомунікаційна компанія оптимізує покриття мережі:

```
Алгоритм AntennaPlacementOptimization(coverage_points, existing_anten
nas):
    // Знайти області без покриття
    all_points = coverage_points
    covered_points = []

    для кожної антени ant в existing_antennas:
        covered = GetCoverageArea(ant)
        covered_points.додати_всі(covered)

    // Знайти опуклу оболонку непокритих точок
    uncovered = SetDifference(all_points, covered_points)
    якщо uncovered порожній:
        повернути []  // Повне покриття

    uncovered_hull = QuickHull(uncovered)

    // Знайти оптимальні позиції для нових антен
    new_positions = []

    поки uncovered не порожній:
        // Знайти центроїд непокритої області
        centroid = CalculateCentroid(uncovered_hull)

        // Знайти найближчу допустиму позицію
        position = FindNearestValidPosition(centroid)
        new_positions.додати(position)

        // Оновити непокриті точки
        new_covered = GetCoverageArea(position)
        uncovered = SetDifference(uncovered, new_covered)

        якщо uncovered не порожній:
            uncovered_hull = QuickHull(uncovered)
```

```
    повернути new_positions
```

Переваги та недоліки

Переваги:

- Інтуїтивність: Алгоритм базується на простій геометричній ідеї “розділяй і володарюй”

- Ефективність на практиці: Середня складність O(n log n) для більшості реальних наборів даних

- Легке узагальнення на 3D: Основні принципи легко переносяться на тривимірний простір

- Ранній відсів точок: Точки всередині трикутників відкидаються одразу, зменшуючи обсяг обчислень

- Паралелізація: Рекурсивна структура дозволяє ефективно розпаралелювати обчислення

- Адаптивність: Продуктивність залежить від фактичної складності опуклої оболонки

Недоліки:

- Квадратична складність у гіршому випадку: $O(n^2)$ для певних конфігурацій точок

- Залежність від вибору початкових точок: Неоптимальний вибір може вплинути на продуктивність

- Глибока рекурсія: Може призвести до переповнення стеку для великих наборів даних

- Числова нестабільність: Обчислення відстаней може давати похибки для близьких точок

- Складність аналізу: Важко передбачити продуктивність для конкретного набору даних

- Необхідність попередньої обробки: Для досягнення оптимальної продуктивності часто потрібна фільтрація точок

Контрольні запитання

1. Поясніть основний принцип роботи алгоритму QuickHull та його зв’язок з QuickSort.

2. Як визначається найвіддаленіша точка від відрізка в алгоритмі?

3. Чому точки всередині трикутника можна безпечно відкидати?

4. Яка часова складність алгоритму в найкращому, середньому та найгіршому випадках?

5. Як алгоритм обробляє колінеарні точки?

6. Опишіть процес рекурсивного поділу в QuickHull.

7. Які оптимізації можна застосувати для покращення продуктивності?

8. Як узагальнити алгоритм для тривимірного простору?

9. Порівняйте QuickHull з алгоритмами Грема та Джарвіса.

10. Які числові проблеми можуть виникнути при реалізації?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму QuickHull для побудови опуклої оболонки

2. Мета

Вивчити принципи роботи алгоритму QuickHull, освоїти техніку “розділяй і володарюй” для геометричних задач, навчитися оптимізувати рекурсивні алгоритми

3. Завдання

- Реалізувати базову версію алгоритму QuickHull

- Створити функції для геометричних обчислень

- Додати візуалізацію процесу побудови

- Реалізувати оптимізації для покращення продуктивності

- Створити версію для цілочисельної арифметики

- Провести порівняльний аналіз з іншими алгоритмами

- Дослідити поведінку на різних типах даних

4. Опис

Лабораторна робота включає повну реалізацію алгоритму QuickHull з візуалізацією кожного етапу рекурсії. Студенти повинні дослідити вплив різних оптимізацій на продуктивність та порівняти результати з іншими алгоритмами побудови опуклої оболонки.

5. Порядок виконання

- Створити базові структури даних для точок та векторів

- Реалізувати функції обчислення відстаней та площ
- Написати функцію визначення положення точки відносно прямої
- Реалізувати функцію перевірки належності точки трикутнику
- Створити рекурсивну функцію FindHull
- Реалізувати основну функцію QuickHull
- Додати попередню фільтрацію точок
- Створити візуалізатор з покроковим виконанням
- Реалізувати версію без рекурсії (з явним стеком)
- Провести тестування та оптимізацію

6. Результат

Програма з графічним інтерфейсом для побудови опуклої оболонки методом QuickHull, яка включає візуалізацію процесу, можливість покрокового виконання, порівняння з іншими алгоритмами та аналіз продуктивності.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
- Модуль геометричних обчислень
- Функція обчислення знакової відстані
- Функція перевірки належності трикутнику
- Рекурсивна функція FindHull
- Основна функція QuickHull
- Модуль попередньої фільтрації
- Ітеративна версія алгоритму
- Візуалізатор з деревом рекурсії
- Модуль тестування та аналізу
- Продемонструвати роботу додатку:
- Покрокове виконання з візуалізацією рекурсії
- Відображення дерева рекурсивних викликів
- Демонстрація відкидання внутрішніх точок
- Робота з різними розподілами точок

- Порівняння часу виконання різних версій
- Аналіз глибини рекурсії для різних даних
- Демонстрація роботи оптимізацій

Бути готовим відповісти на питання:

- Як вибір початкових точок впливає на продуктивність?
- Чому алгоритм має квадратичну складність у гіршому випадку?
- Як реалізувати 3D версію алгоритму?
- Які переваги дає ітеративна версія?
- Як забезпечити числову стабільність?
- Коли QuickHull ефективніший за інші алгоритми?

Тести

1. Яка стратегія використовується в алгоритмі QuickHull?

1) Жадібний алгоритм
2) Динамічне програмування
3) Розділяй і володарюй
4) Повний перебір

2. Що відбувається з точками всередині трикутника в QuickHull?

1) Вони додаються до опуклої оболонки
2) Вони відкидаються
3) Вони обробляються рекурсивно
4) Вони зберігаються для подальшої обробки

3. Яка часова складність QuickHull у середньому випадку?

1) O(n)
2) O(n log n)
3) $O(n^2)$
4) $O(n^3)$

4. Як визначається найвіддаленіша точка від відрізка?

1) За евклідовою відстанню
2) За манхеттенською відстанню
3) За знаковою площею трикутника

4) За кутом

5. Скільки початкових точок використовує алгоритм QuickHull?

1) Одну

2) Дві

3) Три

4) Чотири

Відповіді:

1. 3)

2. 2)

3. 2)

4. 3)

5. 2)

Тести підвищеної складності

1. В якому випадку QuickHull матиме складність $O(n^2)$?

1) Коли точки розподілені випадково

2) Коли всі точки на одній прямій

3) Коли точки утворюють коло

4) Коли точки в кутах квадрата

2. Яка максимальна глибина рекурсії для n точок у найгіршому випадку?

1) O(log n)

2) $O(\sqrt{n})$

3) O(n)

4) $O(n^2)$

3. Як оптимізувати QuickHull для точок на колі?

1) Використати полярні координати

2) Сортувати точки за кутом

3) Використати інший алгоритм

4) Збільшити поріг рекурсії

4. Що станеться, якщо всі точки колінеарні?

1) Алгоритм поверне порожню оболонку

2) Алгоритм поверне дві крайні точки

3) Алгоритм зациклиться

4) Виникне помилка ділення на нуль

5. Яка структура даних найкраще підходить для ітеративної версії?

1) Черга

2) Стек

3) Дерево

4) Хеш-таблиця

6. Як ефективно розпаралелити QuickHull?

1) Паралельно обробляти верхню та нижню частини

2) Паралельно шукати найвіддаленішу точку

3) Паралельно перевіряти належність трикутнику

4) Всі відповіді правильні

7. Яка попередня обробка найбільше прискорить алгоритм?

1) Сортування точок

2) Видалення дублікатів

3) Фільтрація внутрішніх точок

4) Нормалізація координат

8. Для яких даних QuickHull буде найшвидшим серед всіх алгоритмів?

1) Точки на решітці

2) Випадкові точки в крузі

3) Точки з нормальним розподілом

4) Точки на спіралі

9. Як модифікувати QuickHull для побудови альфа-оболонки?

1) Змінити критерій відкидання точок

2) Додати параметр кривизни

3) Використати інший алгоритм

4) Обмежити глибину рекурсії

10. Яка числова проблема найчастіше виникає в QuickHull?

1) Переповнення при множенні

2) Втрата точності при діленні

3) Помилки округлення при порівнянні

4) Всі перераховані

Відповіді

1. 3) Коли точки утворюють коло, кожна рекурсія видаляє лише одну точку

2. 3) У найгіршому випадку глибина рекурсії може досягати O(n)

3. 3) Для точок на колі краще використати інший алгоритм (наприклад, Грема)

4. 2) Алгоритм коректно обробить колінеарні точки і поверне дві крайні

5. 2) Стек найкраще імітує рекурсивні виклики в ітеративній версії

6. 4) Всі згадані аспекти можна ефективно розпаралелити

7. 3) Фільтрація точок всередині опуклої оболонки кутів значно зменшує обсяг даних

8. 3) Для нормального розподілу більшість точок всередині, що дозволяє швидко їх відкинути

9. 3) Альфа-оболонка вимагає принципово іншого підходу, не базованого на опуклості

10. 4) Всі перераховані проблеми можуть виникати при роботі з дійсними числами

Завдання для самостійної роботи

1. Реалізуйте 3D версію алгоритму QuickHull для побудови опуклої оболонки в тривимірному просторі.

2. Розробіть адаптивний алгоритм, який автоматично вибирає між QuickHull, Graham scan та Jarvis march залежно від характеристик вхідних даних.

3. Створіть паралельну версію QuickHull з використанням OpenMP або інших технологій паралельного програмування.

4. Реалізуйте модифікацію QuickHull для роботи з точками на сфері (сферична опукла оболонка).

5. Розробіть інкрементальну версію, яка ефективно оновлює опуклу оболонку при додаванні нових точок.

6. Створіть версію алгоритму з точною арифметикою для уникнення числових помилок.

7. Реалізуйте QuickHull для роботи з зваженими точками (кожна точка має вагу/радіус).

8. Розробіть візуалізатор, який показує залежність продуктивності від розподілу точок.

9. Створіть гібридний алгоритм, який поєднує QuickHull з іншими методами для оптимальної продуктивності.

10. Реалізуйте застосування QuickHull для задачі пошуку діаметра множини точок.

Теми лабораторних робіт

1. Розробка бібліотеки обчислювальної геометрії з реалізацією QuickHull для 2D та 3D просторів.

2. Створення системи аналізу форм 3D об’єктів з використанням опуклих оболонок для спрощення моделей.

3. Реалізація фізичного движка з використанням QuickHull для виявлення колізій між складними об’єктами.

4. Розробка інструменту візуалізації багатовимірних даних через проекції та опуклі оболонки.

5. Створення системи планування траєкторій дронів з урахуванням перешкод через їх опуклі оболонки.

6. Реалізація алгоритму сегментації хмар точок LiDAR з використанням QuickHull для виділення об’єктів.

7. Розробка оптимізатора розміщення об’єктів у просторі з мінімізацією опуклої оболонки.

8. Створення навчального середовища для вивчення алгоритмів обчислювальної геометрії з інтерактивною візуалізацією.

9. Реалізація системи розпізнавання жестів на основі аналізу опуклих оболонок ключових точок.

10. Розробка інструменту для аналізу та оптимізації упаковки об’єктів з використанням опуклих оболонок.

11.2. Задачі близькості: пошук найближчої пари точок, діаграма Вороного та тріангуляція Делоне.

Після того, як ми навчилися окреслювати зовнішні межі набору точок, настав час зазирнути всередину та дослідити їхні внутрішні взаємозв'язки. У цьому розділі ми зосередимося на **задачах близькості**, які відповідають на одне з найбільш фундаментальних питань у просторовому аналізі: "що до чого є найближчим?". Відповіді на це питання є ключовими для кластерного аналізу, розпізнавання образів, планування мереж та багатьох інших прикладних галузей.

Ми вивчимо три потужні концепції, кожна з яких розкриває аспекти близькості з різного боку:

- **Пошук найближчої пари точок:** Це, на перший погляд, проста задача — знайти в мільйонному наборі дві точки, відстань між якими є мінімальною. Хоча наївний перебір усіх пар (O(n2)) є очевидним, ми розглянемо значно ефективніший алгоритм, що базується на принципі "розділяй і володарюй". Він дозволяє вирішити цю задачу за час O(nlogn), що є критично важливим для великих обсягів даних.
- **Діаграма Вороного:** Це одна з найкрасивіших та найпотужніших структур в обчислювальній геометрії. Уявіть, що ви маєте набір "центрів" (лікарень, поштових відділень, мобільних веж). Діаграма Вороного ділить усю площину на "зони впливу" (комірки), де кожна комірка містить усі точки, для яких відповідний центр є найближчим. Ця структура дозволяє миттєво відповідати на запити типу "який об'єкт є найближчим до цієї точки?", що має безліч застосувань у логістиці, містопануванні та біології.
- **Тріангуляція Делоне:** Ця структура є "дзеркальним відображенням" (дуальним графом) діаграми Вороного. Вона з'єднує початкові точки лініями так, щоб утворилася сітка з "якісних" трикутників, які є максимально близькими до рівносторонніх. Головна властивість тріангуляції Делоне — "властивість порожнього кола" — гарантує, що всередині описаного кола будь-якого трикутника сітки немає інших точок. Це робить її незамінною для створення 3D-моделей, інтерполяції даних та моделювання поверхонь.

11.1.3. Алгоритм пошуку найближчої пари точок

Теоретичні відомості

Постановка задачі

Задача пошуку найближчої пари точок є фундаментальною проблемою обчислювальної геометрії. Формально, маючи множину S з n точок на площині, потрібно знайти дві точки p і q з S такі, що евклідова відстань між ними мінімальна серед усіх можливих пар точок.

Евклідова відстань між двома точками $p_1(x_1, y_1)$ та $p_2(x_2, y_2)$ обчислюється за формулою:

$$d(p_1, p_2) = \sqrt{[(x_2 - x_1)^2 + (y_2 - y_1)^2]}$$

Для оптимізації обчислень часто використовується квадрат відстані, що дозволяє уникнути обчислення квадратного кореня:

$$d^2(p_1, p_2) = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

Найпростіший спосіб розв'язання задачі — перевірити всі можливі пари точок:

```
Алгоритм NaiveClosestPair(points):
    Вхід: points - масив з n точок
    Вихід: (p1, p2, min_distance) - найближча пара та відстань між ни
ми

    n = довжина(points)
    min_distance = нескінченність
    closest_p1 = null
    closest_p2 = null

    для i від 0 до n-2:
        для j від i+1 до n-1:
            distance = EuclideanDistance(points[i], points[j])
            якщо distance < min_distance:
                min_distance = distance
                closest_p1 = points[i]
                closest_p2 = points[j]

    повернути (closest_p1, closest_p2, min_distance)

Функція EuclideanDistance(p1, p2):
    dx = p2.x - p1.x
    dy = p2.y - p1.y
    повернути sqrt(dx * dx + dy * dy)
```

Часова складність цього підходу — $O(n^2)$, що неефективно для великих наборів даних.

Алгоритм “розділяй і володарюй”

Ефективний алгоритм пошуку найближчої пари використовує стратегію “розділяй і володарюй” і був розроблений у 1975 році. Основна ідея полягає в рекурсивному поділі множини точок на дві частини та ефективному об’єднанні результатів.

Основні кроки алгоритму:

1. Сортування точок за x-координатою

2. Рекурсивний поділ на ліву та праву половини

3. Знаходження найближчих пар у кожній половині

4. Об’єднання результатів з перевіркою пар між половинами

Розглянемо детальний опис алгоритму.

```
Алгоритм EfficientClosestPair(points):
    Вхід: points - масив з n точок
    Вихід: (p1, p2, distance) - найближча пара та відстань

    // Попереднє сортування
    points_x = SortByX(points)
    points_y = SortByY(points)

    // Виклик рекурсивної функції
    результат = ClosestPairRecursive(points_x, points_y)
    повернути результат

Функція ClosestPairRecursive(px, py):
    Вхід: px - точки, відсортовані за x
           py - точки, відсортовані за y
    Вихід: (p1, p2, distance) - найближча пара

    n = довжина(px)

    // Базовий випадок: мало точок
    якщо n <= 3:
        повернути NaiveClosestPair(px)

    // Поділ на дві половини
    mid = n / 2
    midpoint = px[mid]
```

```
    // Розділення py на ліву та праву частини
    pyl = []  // точки зліва від midpoint
    pyr = []  // точки справа від midpoint

    для кожної точки p в py:
        якщо p.x <= midpoint.x:
            pyl.додати(p)
        інакше:
            pyr.додати(p)

    // Рекурсивні виклики
    (p1_left, p2_left, dl) = ClosestPairRecursive(px[0:mid], pyl)
    (p1_right, p2_right, dr) = ClosestPairRecursive(px[mid:n], pyr)

    // Знаходження мінімальної відстані
    якщо dl < dr:
        d = dl
        closest_pair = (p1_left, p2_left)
    інакше:
        d = dr
        closest_pair = (p1_right, p2_right)

    // Перевірка пар між половинами
    strip = []
    для кожної точки p в py:
        якщо abs(p.x - midpoint.x) < d:
            strip.додати(p)

    // Знаходження найближчої пари в смузі
    (p1_strip, p2_strip, d_strip) = ClosestInStrip(strip, d)

    якщо d_strip < d:
        повернути (p1_strip, p2_strip, d_strip)
    інакше:
        повернути (closest_pair[0], closest_pair[1], d)
```

Ключова оптимізація: обробка смуги

Найважливіша частина алгоритму — ефективна перевірка пар точок між лівою та правою половинами. Ключове спостереження: якщо найближча пара має одну точку в лівій половині та одну в правій, то обидві точки повинні знаходитися в смузі шириною 2d навколо розділяючої лінії.

```
Функція ClosestInStrip(strip, d):
    Вхід: strip - точки в смузі, відсортовані за y
```

```
        d - поточна мінімальна відстань
    Вихід: найближча пара в смузі

    min_distance = d
    closest_p1 = null
    closest_p2 = null
    n = довжина(strip)

    // Для кожної точки перевіряємо лише кілька наступних
    для i від 0 до n-1:
        j = i + 1
        // Перевіряємо лише точки з різницею y-координат < d
        поки j < n та (strip[j].y - strip[i].y) < d:
            distance = EuclideanDistance(strip[i], strip[j])
            якщо distance < min_distance:
                min_distance = distance
                closest_p1 = strip[i]
                closest_p2 = strip[j]
            j = j + 1

    повернути (closest_p1, closest_p2, min_distance)
```

Лема про смугу: Якщо точки в смузі відсортовані за y-координатою, то для кожної точки достатньо перевірити не більше 7 наступних точок.

Доведення: Розглянемо прямокутник розміром d × 2d, розділений вертикальною лінією посередині. В кожній половині прямокутника (розміром d × d) не може бути більше 4 точок, відстань між якими ⩾ d. Отже, всього в прямокутнику може бути максимум 8 точок, і нам потрібно перевірити максимум 7 пар.

Аналіз складності

Часова складність:

- Початкове сортування: O(n log n)

- Рекурсивні виклики: T(n) = 2T(n/2) + O(n)

- За основною теоремою: T(n) = O(n log n)

Просторова складність: O(n) для зберігання проміжних масивів.

Розглянемо набір з 8 точок:

```
Точки: A(2,3), B(5,4), C(8,2), D(9,6), E(11,3), F(12,7), G(14,5), H(15,8)
```

Крок 1: Сортування

```
За x: A(2,3), B(5,4), C(8,2), D(9,6), E(11,3), F(12,7), G(14,5), H(15
,8)
За y: C(8,2), A(2,3), E(11,3), B(5,4), G(14,5), D(9,6), F(12,7), H(15
,8)
```

Візуалізація початкового стану:

```
  8 |                                  H
  7 |                        F
  6 |             D
  5 |                               G
  4 |     B
  3 | A                E
  2 |         C
  1 |
  0 +--------------------------------
    0 2   5   8 9   11 12   14 15
```

Крок 2: Перший поділ

Середина: між D та E - Ліва половина: A, B, C, D - Права половина: E, F, G, H

Крок 3: Рекурсивна обробка лівої половини

Поділ лівої половини: - Ліва: A, B - Права: C, D

Найближчі пари: - У лівій частині: (A, B) з відстанню $\sqrt{[(5-2)^2 + (4-3)^2]} = \sqrt{10} \approx 3.16$ - У правій частині: (C, D) з відстанню $\sqrt{[(9-8)^2 + (6-2)^2]} = \sqrt{17} \approx 4.12$

Крок 4: Обробка смуги між A,B та C,D

Смуга навколо x = 6.5 з шириною 2×3.16: - Точки в смузі: B(5,4), C(8,2) - Відстань B-C: $\sqrt{[(8-5)^2 + (2-4)^2]} = \sqrt{13} \approx 3.61$

Мінімум для лівої половини: (A, B) з відстанню 3.16

Крок 5: Аналогічно для правої половини

Результат: (E, G) з відстанню $\sqrt{[(14-11)^2 + (5-3)^2]} = \sqrt{13} \approx 3.61$

Крок 6: Фінальна смуга

Смуга між лівою та правою половинами (навколо x = 10): - Точки в смузі з $|x - 10| < 3.16$: D(9,6), E(11,3) - Відстань D-E: $\sqrt{[(11-9)^2 + (3-6)^2]} = \sqrt{13} \approx 3.61$

Фінальний результат: Найближча пара - (A, B) з відстанню $\sqrt{10} \approx 3.16$

Оптимізації алгоритму

1. Використання квадратів відстаней

Для уникнення обчислення квадратних коренів:

```
Функція DistanceSquared(p1, p2):
    dx = p2.x - p1.x
```

```
    dy = p2.y - p1.y
    повернути dx * dx + dy * dy
```
2. Попередня фільтрація

Видалення дублікатів точок перед початком алгоритму:

```
Функція RemoveDuplicates(points):
    unique_points = []
    point_set = новий набір

    для кожної точки p в points:
        якщо p не в point_set:
            point_set.додати(p)
            unique_points.додати(p)

    повернути unique_points
```
Область застосування

Алгоритм пошуку найближчої пари точок має широкий спектр застосувань:

- Комп'ютерна графіка: Виявлення колізій між об'єктами, оптимізація рендерингу, спрощення полігональних моделей

- Кластерний аналіз: Ініціалізація алгоритмів кластеризації, визначення щільності кластерів, виявлення аномалій

- Географічні інформаційні системи: Пошук найближчих об'єктів інфраструктури, аналіз просторового розподілу, планування маршрутів

- Астрономія: Виявлення подвійних зірок, аналіз розподілу галактик, пошук гравітаційних взаємодій

- Біоінформатика: Аналіз просторової структури білків, виявлення схожих генетичних послідовностей

- Управління повітряним рухом: Запобігання зіткненням літаків, оптимізація маршрутів польотів

- Робототехніка: Навігація в просторі, уникнення перешкод, планування руху маніпуляторів

Приклади застосування

Приклад 1: Система попередження зіткнень для дронів

```
Алгоритм DroneCollisionWarning(drones, safety_distance):
    Вхід: drones - масив позицій дронів
           safety_distance - мінімальна безпечна відстань
    Вихід: warnings - список попереджень
```

```
    warnings = []

    // Знаходимо найближчу пару
    (drone1, drone2, distance) = EfficientClosestPair(drones)

    якщо distance < safety_distance:
        warnings.додати({
            drones: [drone1, drone2],
            distance: distance,
            priority: "CRITICAL"
        })

    // Знаходимо всі небезпечні пари
    dangerous_pairs = FindAllPairsCloserThan(drones, safety_distance
* 1.5)

    для кожної пари (d1, d2, dist) в dangerous_pairs:
        якщо dist < safety_distance * 1.2:
            priority = "HIGH"
        інакше:
            priority = "MEDIUM"

        warnings.додати({
            drones: [d1, d2],
            distance: dist,
            priority: priority
        })

    повернути warnings

Функція FindAllPairsCloserThan(points, threshold):
    // Модифікована версія алгоритму для знаходження всіх близьких па
р
    результат = []
    // Використовуємо просторове індексування для ефективності
    grid = SpatialGrid(points, threshold)

    для кожної точки p в points:
        neighbors = grid.GetNeighbors(p, threshold)
        для кожного сусіда q в neighbors:
            якщо p < q:  // уникаємо дублікатів
                dist = EuclideanDistance(p, q)
                якщо dist < threshold:
                    результат.додати((p, q, dist))
```

```
    повернути результат
```

Приклад 2: Оптимізація розміщення базових станцій

```
Алгоритм OptimizeBaseStations(stations, coverage_radius):
    Вхід: stations - існуючі базові станції
           coverage_radius - радіус покриття
    Вихід: recommendations - рекомендації щодо оптимізації

    recommendations = []

    // Знаходимо найближчі станції
    (s1, s2, distance) = EfficientClosestPair(stations)

    якщо distance < coverage_radius * 0.5:
        // Станції занадто близько - неефективне покриття
        recommendations.додати({
            type: "REDUNDANCY",
            stations: [s1, s2],
            action: "Розглянути об'єднання або переміщення"
        })

    // Аналіз покриття
    coverage_gaps = AnalyzeCoverageGaps(stations, coverage_radius)

    для кожного gap в coverage_gaps:
        nearest = FindNearestStation(gap.center, stations)
        якщо відстань(gap.center, nearest) > coverage_radius * 2:
            recommendations.додати({
                type: "GAP",
                location: gap.center,
                action: "Встановити нову станцію"
            })

    повернути recommendations
```

Приклад 3: Кластеризація точок продажу

```
Алгоритм ClusterSalesPoints(sales_points, max_cluster_size):
    Вхід: sales_points - точки продажу з координатами
           max_cluster_size - максимальний розмір кластера
    Вихід: clusters - список кластерів

    unassigned = копія(sales_points)
    clusters = []

    поки unassigned не порожній:
```

```
        // Знаходимо найщільнішу область
        якщо довжина(unassigned) >= 2:
            (p1, p2, distance) = EfficientClosestPair(unassigned)

            // Створюємо новий кластер
            cluster = новий Кластер()
            cluster.додати(p1)
            cluster.додати(p2)
            unassigned.видалити(p1)
            unassigned.видалити(p2)

            // Розширюємо кластер
            changed = true
            поки changed та cluster.розмір() < max_cluster_size:
                changed = false
                nearest = null
                min_dist = нескінченність

                для кожної точки p в unassigned:
                    для кожної точки c в cluster:
                        dist = EuclideanDistance(p, c)
                        якщо dist < min_dist:
                            min_dist = dist
                            nearest = p

                якщо min_dist < distance * 2:  // поріг розширення
                    cluster.додати(nearest)
                    unassigned.видалити(nearest)
                    changed = true

            clusters.додати(cluster)
        інакше:
            // Залишилася одна точка
            cluster = новий Кластер()
            cluster.додати(unassigned[0])
            clusters.додати(cluster)
            unassigned.очистити()

    повернути clusters
```

Переваги та недоліки

Переваги:

- Оптимальна часова складність: O(n log n) є теоретично оптимальною для цієї задачі

- Гарантована точність: Алгоритм завжди знаходить точну найближчу пару
- Масштабованість: Ефективно працює з великими наборами даних
- Універсальність: Легко адаптується для різних метрик відстані
- Паралелізація: Рекурсивна структура дозволяє ефективне розпаралелювання
- Розширюваність: Можна модифікувати для пошуку k найближчих пар

Недоліки:

- Складність реалізації: Вимагає акуратної обробки граничних випадків
- Накладні витрати: Для малих n наївний алгоритм може бути швидшим
- Використання пам’яті: Потребує O(n) додаткової пам’яті для проміжних масивів
- Чутливість до розмірності: У високих розмірностях ефективність знижується
- Попереднє сортування: Вимагає часу O(n log n) навіть якщо точки вже частково впорядковані
- Складність налагодження: Рекурсивна природа ускладнює відлагодження

Контрольні запитання

1. Чому наївний алгоритм має складність $O(n^2)$?
2. Поясніть основну ідею алгоритму “розділяй і володарюй” для цієї задачі.
3. Чому достатньо перевірити лише 7 точок у смузі?
4. Як змінити алгоритм для використання манхеттенської відстані?
5. Чи можна покращити складність O(n log n)?
6. Як обробляються дублікати точок?
7. Яка роль сортування в алгоритмі?
8. Як алгоритм працює в тривимірному просторі?
9. Які оптимізації можливі для спеціальних випадків розподілу точок?
10. Як знайти k найближчих пар замість однієї?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму пошуку найближчої пари точок

2. Мета

Вивчити принципи роботи алгоритму “розділяй і володарюй”, навчитися ефективно розв’язувати геометричні задачі близькості, освоїти техніки оптимізації рекурсивних алгоритмів

3. Завдання

- Реалізувати наївний алгоритм пошуку найближчої пари
- Реалізувати ефективний алгоритм “розділяй і володарюй”
- Створити візуалізацію роботи алгоритму
- Провести порівняльний аналіз продуктивності
- Реалізувати варіанти для різних метрик відстані
- Розширити алгоритм для пошуку k найближчих пар
- Оптимізувати для спеціальних випадків

4. Опис

Лабораторна робота передбачає повну реалізацію алгоритму пошуку найближчої пари точок з візуалізацією процесу рекурсивного поділу та об’єднання результатів. Особлива увага приділяється ефективній обробці смуги між половинами.

5. Порядок виконання

- Створити структури даних для точок
- Реалізувати функції обчислення відстаней
- Написати наївний алгоритм для порівняння
- Реалізувати функції сортування точок
- Створити рекурсивну функцію поділу
- Реалізувати ефективну обробку смуги
- Додати візуалізацію з покроковим виконанням
- Створити генератори тестових даних
- Провести вимірювання продуктивності
- Реалізувати додаткові оптимізації

6. Результат

Програма з графічним інтерфейсом для пошуку найближчої пари точок, яка демонструє роботу алгоритму, дозволяє порівнювати різні підходи та аналізувати їх ефективність на різних типах даних.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
- Модуль роботи з точками та відстанями
- Наївний алгоритм $O(n^2)$
- Функції сортування за координатами
- Рекурсивна функція поділу
- Функція обробки смуги
- Модуль візуалізації
- Генератори тестових даних
- Модуль вимірювання продуктивності
- Продемонструвати роботу додатку:
- Покрокове виконання алгоритму
- Візуалізація рекурсивних поділів
- Підсвічування поточної смуги
- Порівняння з наївним алгоритмом
- Робота з різними розподілами точок
- Графіки залежності часу від розміру даних
- Демонстрація оптимізацій

Бути готовим відповісти на питання:

- Чому перевірка 7 точок у смузі достатня?
- Як довести коректність алгоритму?
- Які випадки найгірші для продуктивності?
- Як адаптувати для інших метрик?
- Чи можна покращити константу в $O(n \log n)$?
- Як працювати з точками на прямій?

Тести

1. Яка часова складність наївного алгоритму пошуку найближчої пари?

1) $O(n)$

2) $O(n \log n)$

3) $O(n^2)$

4) $O(n^3)$

2. Скільки максимально точок потрібно перевірити в смузі для кожної точки?

1) Всі точки в смузі

2) 7 точок

3) $\log n$ точок

4) $\sqrt{n}$ точок

3. Яка стратегія використовується в ефективному алгоритмі?

1) Динамічне програмування

2) Жадібний алгоритм

3) Розділяй і володарюй

4) Повний перебір

4. Що є ключовою оптимізацією алгоритму?

1) Використання хеш-таблиць

2) Ефективна обробка смуги

3) Паралельне виконання

4) Кешування результатів

5. Яка часова складність оптимального алгоритму?

1) $O(n)$

2) $O(n \log n)$

3) $O(n \log^2 n)$

4) $O(n\sqrt{n})$

Відповіді:

1. 3)

2. 2)

3. 3)

4. 2)

5. 2)

Тести підвищеної складності

1. Для точок на прямій лінії алгоритм матиме складність:

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) $O(n \log^2 n)$

2. При використанні манхеттенської відстані замість евклідової:

1) Алгоритм не працюватиме

2) Потрібно змінити константу 7 на іншу

3) Складність стане $O(n^2)$

4) Алгоритм працюватиме без змін

3. Для знаходження k найближчих пар складність становитиме:

1) O(kn log n)

2) O(n log n + k)

3) O(n log n + k log k)

4) $O(kn^2)$

4. У тривимірному просторі максимальна кількість точок для перевірки в “смузі”:

1) 7

2) 15

3) 27

4) Константа, але більша за 7

5. Якщо всі точки мають однакову x-координату:

1) Алгоритм зациклиться

2) Складність буде $O(n^2)$

3) Алгоритм працюватиме за O(n log n)

4) Виникне помилка поділу

6. Оптимальний розмір для переходу на наївний алгоритм:

1) 3 точки

2) 10-20 точок

3) 100 точок

4) $\sqrt{n}$ точок

7. При попередньому видаленні дублікатів:

1) Алгоритм може дати неправильний результат

2) Покращиться лише константа

3) Складність стане O(n)

4) Алгоритм не зміниться

8. Для рівномірно розподілених точок у квадраті очікувана відстань:

1) $O(1/\sqrt{n})$

2) O(1/n)

3) $O(\sqrt{n})$

4) O(log n/n)

9. Використання k-d дерева замість сортування дасть складність:

1) O(n log n) в середньому

2) O(n) в середньому

3) $O(n^2)$ в гіршому випадку

4) Обидві відповіді a) та c) правильні

10. Для точок на колі найгірший випадок виникає коли:

1) Точки розподілені рівномірно

2) Точки згруповані в одному секторі

3) Є дві групи точок

4) Точки розташовані випадково

Відповіді:

1. 2) Алгоритм все одно матиме складність O(n log n) через необхідність сортування

2. 4) Основна структура алгоритму не залежить від конкретної метрики

3. 3) Потрібно знайти та відсортувати k найменших відстаней

4. 4) У 3D константа більша, приблизно 12-15 точок

5. 3) Алгоритм коректно обробить цей випадок зі стандартною складністю

6. 2) На практиці 10-20 точок оптимальні для переходу на наївний метод

7. 2) Видалення дублікатів лише зменшить n, не змінюючи асимптотику

8. 1) Для рівномірного розподілу очікувана найближча відстань ~ $O(1/\sqrt{n})$

9. 4) k-d дерево дає $O(n \log n)$ в середньому, але $O(n^2)$ в гіршому випадку

10. 1) Рівномірний розподіл на колі створює багато пар з однаковими відстанями

Завдання для самостійної роботи

1. Реалізуйте алгоритм пошуку найближчої пари для точок у k-вимірному просторі та дослідіть залежність продуктивності від розмірності.

2. Розробіть динамічну версію алгоритму, яка ефективно обробляє додавання та видалення точок.

3. Створіть паралельну версію алгоритму з використанням OpenMP або MPI.

4. Реалізуйте алгоритм для різних метрик: манхеттенська, Чебишова, Мінковського.

5. Розробіть модифікацію для пошуку найвіддаленішої пари точок.

6. Створіть алгоритм пошуку всіх пар точок з відстанню менше заданого порогу.

7. Реалізуйте версію з використанням просторових структур даних (k-d дерево, R-дерево).

8. Розробіть алгоритм для точок з вагами, де потрібно мінімізувати зважену відстань.

9. Створіть адаптивний алгоритм, який вибирає оптимальну стратегію залежно від розподілу точок.

10. Реалізуйте застосування для задачі кластеризації з автоматичним визначенням кількості кластерів.

Теми лабораторних робіт

1. Розробка системи моніторингу повітряного простору з виявленням потенційно небезпечних зближень.

2. Створення інструменту аналізу молекулярних структур для виявлення взаємодіючих атомів.

3. Реалізація системи оптимізації розміщення точок обслуговування в містах.

4. Розробка алгоритму стиснення зображень на основі видалення близьких кольорів.

5. Створення системи розпізнавання дублікатів у великих базах геоданих.

6. Реалізація інтелектуальної системи групування товарів у складських приміщеннях.

7. Розробка інструменту для аналізу соціальних мереж та виявлення тісно пов’язаних груп.

8. Створення системи автоматичного розміщення міток на картах без перекриття.

9. Реалізація алгоритму оптимізації маршрутів доставки з урахуванням близькості точок.

10. Розробка системи виявлення аномалій у часових рядах на основі аналізу близькості.

11.3. Перетини та належність: перетин відрізків (Bentley-Ottmann), точка в багатокутнику (Ray casting).

Задачі визначення геометричних перетинів та належності є фундаментальними в обчислювальній геометрії та мають критичне значення для багатьох практичних застосувань. Ці задачі виникають щоразу, коли потрібно визначити взаємне розташування геометричних об’єктів: чи перетинаються вони, чи містить один об’єкт інший, чи знаходиться точка всередині або зовні заданої області. Від ефективності розв’язання цих задач залежить продуктивність систем комп’ютерної графіки, геоінформаційних систем, САПР та багатьох інших технологій.

Визначення перетинів між відрізками — одна з найпоширеніших операцій в геометричних алгоритмах. На перший погляд, задача здається простою: маючи два відрізки, потрібно визначити, чи перетинаються вони. Однак коли мова йде про велику кількість відрізків — сотні, тисячі або мільйони — наївний підхід перевірки всіх пар стає неприйнятно повільним.

Уявіть карту міста з тисячами вулиць, де кожна вулиця представлена відрізком. Для планування маршрутів, аналізу транспортних потоків або визначення перехресть потрібно знати всі точки перетину. Наївний алгоритм, що перевіряє кожну пару вулиць, мав би складність $O(n^2)$, що для 10,000 вулиць означало б 100 мільйонів перевірок.

Алгоритм Bentley-Ottmann, розроблений у 1979 році, елегантно вирішує цю проблему, використовуючи техніку заметання площини (plane sweep). Замість перевірки всіх пар, алгоритм "заметає" площину вертикальною лінією зліва направо, відстежуючи лише ті відрізки, які можуть перетинатися. Це дозволяє досягти складності $O((n + k) \log n)$, де k — кількість перетинів, що значно ефективніше для практичних випадків, коли $k << n^2$.

Друга фундаментальна задача — **визначення, чи знаходиться точка всередині багатокутника** — має не менш широке коло застосувань. Від простої перевірки, чи клікнув користувач на певному об'єкті інтерфейсу, до складних геопросторових запитів типу "в якій країні знаходиться дана GPS-координата" — всюди потрібне ефективне розв'язання цієї задачі.

Алгоритм променевого сканування (ray casting), який ми детально розглянемо, базується на простій, але потужній ідеї: якщо провести промінь з точки в довільному напрямку, то кількість перетинів цього променя з межею багатокутника визначає положення точки. Парна кількість перетинів означає, що точка зовні, непарна — всередині. Ця елегантна ідея, однак, потребує акуратної реалізації для обробки особливих випадків: що робити, якщо промінь проходить через вершину багатокутника або співпадає з його стороною?

Ефективне розв'язання задач перетину та належності критично важливе для багатьох галузей:

Комп'ютерна графіка та ігрова індустрія використовують ці алгоритми для рендерингу сцен, визначення видимості об'єктів, обробки тіней та відсікання невидимих поверхонь. Кожен кадр сучасної 3D-гри потребує мільйонів таких перевірок.

Геоінформаційні системи застосовують алгоритми перетину для аналізу топології карт, пошуку перехресть доріг, визначення меж територій. Запити типу "знайти всі заправки в межах міста" безпосередньо використовують алгоритми належності точки багатокутнику.

Системи автоматизованого проектування (САПР) потребують точного визначення перетинів для перевірки коректності креслень, виявлення колізій між деталями, оптимізації розкрою матеріалів.

Робототехніка та автономні системи використовують ці алгоритми для планування траєкторій, уникнення перешкод, визначення досяжних зон. Безпілотний автомобіль повинен миттєво визначати, чи перетинає запланована траєкторія межі дороги або перешкоди.

Незважаючи на концептуальну простоту, практична реалізація алгоритмів перетину та належності стикається з численними викликами:

Числова точність стає критичною, коли відрізки майже паралельні або точка знаходиться дуже близько до межі багатокутника. Помилки округлення можуть призвести до неправильних результатів або навіть до збоїв програми.

Вироджені випадки — ситуації, коли об’єкти мають спеціальне взаємне розташування (співпадаючі кінцеві точки, колінеарні відрізки, точка на межі багатокутника) — потребують особливої уваги та окремої обробки.

Ефективність для великих даних вимагає не лише правильного вибору алгоритму, але й оптимальних структур даних, балансування між попередньою обробкою та часом запиту, можливо, використання паралельних обчислень.

Історично, задачі перетину та належності розв’язувалися простими методами з квадратичною або вищою складністю. З розвитком комп’ютерної графіки та збільшенням обсягів даних виникла потреба в більш ефективних алгоритмах.

Техніка заметання площини, вперше систематично застосована в алгоритмі Bentley-Ottmann, стала революційною, показавши, як можна звести двовимірну задачу до послідовності одновимірних. Ця ідея згодом була розвинена та застосована до багатьох інших геометричних задач.

Для задачі належності точки багатокутнику також було розроблено кілька підходів: від простого підрахунку кутів до методу променевого сканування та алгоритму на основі числа обертання. Кожен має свої переваги та недоліки, і вибір залежить від конкретних вимог застосування.

У цьому підрозділі ми детально розглянемо класичні та сучасні підходи до розв’язання задач перетину та належності, вивчимо їх теоретичні основи, практичні аспекти реалізації та численні застосування.

11.3.1. Алгоритм перетину відрізків Бентлі-Оттманна

Теоретичні відомості

Постановка задачі

Задача пошуку всіх перетинів у множині відрізків формулюється так: маючи набір з n відрізків на площині, знайти всі точки, де принаймні два відрізки перетинаються. Ця задача є фундаментальною в обчислювальній геометрії та має численні практичні застосування.

Наївний підхід полягав би в перевірці кожної пари відрізків на перетин, що дає складність $O(n^2)$. Для великих наборів даних це неприйнятно повільно. Алгоритм Бентлі-Оттманна, розроблений Джоном Бентлі та Томасом Оттманном у 1979 році, використовує техніку заметання площини для досягнення складності $O((n + k) \log n)$, де k — кількість перетинів.

Основна ідея: заметання площини

Техніка заметання площини — це парадигма розробки геометричних алгоритмів, яка полягає в уявному переміщенні вертикальної лінії (лінії заметання) через площину зліва направо. Під час руху лінії відстежуються зміни в перетині цієї лінії з геометричними об’єктами.

Уявіть вертикальну лінію, яка рухається зліва направо через площину з відрізками. В кожен момент часу ця лінія перетинає деякі відрізки. Ключове спостереження: якщо два відрізки перетинаються в точці P, то безпосередньо перед проходженням лінії заметання через P ці відрізки будуть сусідніми в порядку їх перетину з лінією заметання (впорядкованими за y-координатою).

Алгоритм використовує дві основні структури даних:

Черга подій (Event Queue) — пріоритетна черга, що містить події, впорядковані за x-координатою. Події бувають трьох типів: - Початок відрізка (лівий кінець) — коли лінія заметання досягає лівого кінця відрізка - Кінець відрізка (правий кінець) — коли лінія заметання залишає відрізок - Перетин двох відрізків — коли лінія заметання проходить через точку перетину

Стан заметання (Sweep Line Status) — збалансоване бінарне дерево пошуку, що зберігає відрізки, які перетинають поточну лінію заметання, впорядковані за y-координатою точки перетину. Це дерево динамічно оновлюється під час руху лінії заметання.

Наведемо детальний опис алгоритму.

```
Алгоритм BentleyOttmann(segments):
    Вхід: segments - масив відрізків
    Вихід: intersections - множина точок перетину

    // Ініціалізація структур даних
    // event_queue зберігатиме всі події впорядковані за x-координато
ю
    event_queue = новий ПріоритетнаЧерга()

    // sweep_status зберігатиме відрізки, що перетинають поточну ліні
ю заметання
    sweep_status = новий ЗбалансованеДерево()

    // intersections накопичуватиме знайдені точки перетину
    intersections = новий Набір()

    // Додаємо початкові події - кінці всіх відрізків
    для кожного відрізка s в segments:
        p1 = s.початок
        p2 = s.кінець

        // Впорядковуємо кінці відрізка так, щоб p1 був лівішим
        // Це важливо для коректної роботи алгоритму
        якщо p1.x > p2.x або (p1.x == p2.x та p1.y > p2.y):
            обміняти(p1, p2)

        // Додаємо дві події для кожного відрізка
        event_queue.додати(Подія(p1, START_EVENT, s))
        event_queue.додати(Подія(p2, END_EVENT, s))

    // Основний цикл - обробляємо події в порядку їх x-координат
    поки не event_queue.порожня():
        подія = event_queue.вилучити_мін()

        // Обробляємо подію залежно від її типу
        якщо подія.тип == START_EVENT:
            HandleStartEvent(подія, sweep_status, event_queue, inters
ections)
        інакше якщо подія.тип == END_EVENT:
            HandleEndEvent(подія, sweep_status, event_queue, intersec
tions)
        інакше: // INTERSECTION_EVENT
            HandleIntersectionEvent(подія, sweep_status, event_queue,
```

```
intersections)

    повернути intersections
```

Обробка подій

Обробка початку відрізка:

Коли лінія заметання досягає початку відрізка, цей відрізок потрібно додати до стану заметання та перевірити можливі перетини з його новими сусідами.

```
Процедура HandleStartEvent(event, status, queue, intersections):
    segment = event.segment

    // Вставляємо відрізок у стан заметання
    // Позиція визначається у-координатою на поточній х-позиції
    status.вставити(segment)

    // Після вставки знаходимо сусідів нового відрізка
    // above - відрізок безпосередньо вище в поточному стані
    above = status.наступник(segment)
    // below - відрізок безпосередньо нижче в поточному стані
    below = status.попередник(segment)

    // Перевіряємо можливі перетини з сусідами
    // Важливо: перевіряємо лише з безпосередніми сусідами!
    якщо above != null:
        CheckIntersection(segment, above, event.point.x, queue)

    якщо below != null:
        CheckIntersection(below, segment, event.point.x, queue)
```

Обробка кінця відрізка:

Коли лінія заметання залишає відрізок, його потрібно видалити зі стану. При цьому його колишні сусіди стають сусідами один одного.

```
Процедура HandleEndEvent(event, status, queue, intersections):
    segment = event.segment

    // Спочатку знаходимо сусідів відрізка, що видаляється
    above = status.наступник(segment)
    below = status.попередник(segment)

    // Видаляємо відрізок зі стану заметання
    status.видалити(segment)

    // Після видалення above та below стали сусідами
```

```
    // Перевіряємо, чи перетинаються вони
    якщо above != null та below != null:
        CheckIntersection(below, above, event.point.x, queue)
```

Обробка перетину:

Це найскладніша частина алгоритму. Коли два відрізки перетинаються, їх відносний порядок у стані заметання змінюється.

```
Процедура HandleIntersectionEvent(event, status, queue, intersections
):
    s1 = event.segment1
    s2 = event.segment2

    // Зберігаємо точку перетину в результаті
    intersections.додати(event.point)

    // До перетину s1 був вище s2, після перетину - навпаки
    // Тому потрібно оновити їх порядок у дереві стану

    // Спочатку видаляємо обидва відрізки
    status.видалити(s1)
    status.видалити(s2)

    // Вставляємо їх назад у зміненому порядку
    // Дерево автоматично розмістить їх правильно
    status.вставити(s2)
    status.вставити(s1)

    // Після зміни порядку у відрізків змінилися сусіди
    // Перевіряємо нові сусідні пари на перетини
    above_s1 = status.наступник(s1)
    below_s2 = status.попередник(s2)

    якщо above_s1 != null:
        CheckIntersection(s1, above_s1, event.point.x, queue)

    якщо below_s2 != null:
        CheckIntersection(below_s2, s2, event.point.x, queue)
```

Перевірка перетину та додавання події

```
Процедура CheckIntersection(s1, s2, current_x, queue):
    // Обчислюємо точку перетину двох відрізків
    intersection = ComputeIntersection(s1, s2)

    якщо intersection != null:
        // Важливо: додаємо подію перетину лише якщо вона
```

```
        // відбувається праворуч від поточної позиції лінії заметання
        // Це запобігає обробці однієї події кілька разів
        якщо intersection.x > current_x або
           (intersection.x == current_x та intersection не в queue):
            queue.додати(Подія(intersection, INTERSECTION_EVENT, s1,
s2))

Функція ComputeIntersection(s1, s2):
    // Обчислюємо перетин двох відрізків використовуючи параметричні
рівняння
    // Відрізок можна представити як P(t) = P1 + t(P2 - P1), де t ∈ [
0,1]

    // Координати кінців першого відрізка
    x1 = s1.початок.x, y1 = s1.початок.y
    x2 = s1.кінець.x, y2 = s1.кінець.y

    // Координати кінців другого відрізка
    x3 = s2.початок.x, y3 = s2.початок.y
    x4 = s2.кінець.x, y4 = s2.кінець.y

    // Обчислюємо знаменник для формул перетину
    // Якщо знаменник = 0, відрізки паралельні
    denom = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4)

    якщо abs(denom) < EPSILON:
        // Відрізки паралельні або колінеарні
        // Для спрощення вважаємо, що вони не перетинаються
        повернути null

    // Обчислюємо параметри t та u для точки перетину
    // t - параметр для першого відрізка
    t = ((x1 - x3) * (y3 - y4) - (y1 - y3) * (x3 - x4)) / denom
    // u - параметр для другого відрізка
    u = -((x1 - x2) * (y1 - y3) - (y1 - y2) * (x1 - x3)) / denom

    // Перетин існує лише якщо обидва параметри в діапазоні [0,1]
    якщо 0 <= t <= 1 та 0 <= u <= 1:
        // Обчислюємо координати точки перетину
        x = x1 + t * (x2 - x1)
        y = y1 + t * (y2 - y1)
        повернути Точка(x, y)

    повернути null
```

Порівняння відрізків у стані заметання

Важливою частиною алгоритму є функція порівняння відрізків для їх впорядкування в дереві стану:

```
Функція CompareSegments(s1, s2, x):
    // Порівнює два відрізки за їх y-координатою на вертикалі x
    // Повертає -1 якщо s1 нижче s2, 1 якщо вище, 0 якщо рівні

    y1 = GetYAtX(s1, x)
    y2 = GetYAtX(s2, x)

    якщо y1 < y2:
        повернути -1
    інакше якщо y1 > y2:
        повернути 1
    інакше:
        // Якщо y-координати рівні, порівнюємо за нахилом
        // Це важливо для коректної обробки дотичних відрізків
        slope1 = (s1.кінець.y - s1.початок.y) / (s1.кінець.x - s1.поч
аток.x)
        slope2 = (s2.кінець.y - s2.початок.y) / (s2.кінець.x - s2.поч
аток.x)

        якщо slope1 < slope2:
            повернути -1
        інакше якщо slope1 > slope2:
            повернути 1
        інакше:
            повернути 0

Функція GetYAtX(segment, x):
    // Обчислює y-координату відрізка на вертикалі x
    // Використовує лінійну інтерполяцію

    x1 = segment.початок.x
    y1 = segment.початок.y
    x2 = segment.кінець.x
    y2 = segment.кінець.y

    якщо x1 == x2:
        // Вертикальний відрізок
        повернути y1

    // Лінійна інтерполяція
    t = (x - x1) / (x2 - x1)
    повернути y1 + t * (y2 - y1)
```

Розглянемо детально роботу алгоритму на наборі з 4 відрізків:

```
S1: від (1, 2) до (5, 4)
S2: від (2, 5) до (4, 1)
S3: від (3, 3) до (6, 3)
S4: від (1, 4) до (5, 2)
```

Візуалізація початкового стану:

```
  5 |  S2\
  4 |S4  \   S1/
  3 |  \  X / S3----
  2 | S1\ X S4/
  1 |    \S2/
  0 +---------------
    0 1 2 3 4 5 6
```

Крок 1: Ініціалізація

Створюємо чергу подій з усіх кінців відрізків:

```
Події (впорядковані за x):
1. (1, 2) - початок S1
2. (1, 4) - початок S4
3. (2, 5) - початок S2
4. (3, 3) - початок S3
5. (4, 1) - кінець S2
6. (5, 2) - кінець S4
7. (5, 4) - кінець S1
8. (6, 3) - кінець S3
```

Крок 2: Обробка події x = 1

Лінія заметання на x = 1. Обробляємо початок S1 та S4:

```
Подія: початок S1 (1, 2)
- Додаємо S1 до стану
- Стан: [S1]
- Сусідів немає, перевірок немає

Подія: початок S4 (1, 4)
- Додаємо S4 до стану
- Стан: [S1, S4] (S1 нижче, бо y=2 < y=4)
- S1 та S4 - сусіди
- Перевіряємо перетин S1-S4
- Знайдено перетин у точці (3, 3)
- Додаємо подію перетину до черги
```

Крок 3: Обробка події x = 2

```
Подія: початок S2 (2, 5)
- На x=2: S1 має y≈2.5, S4 має y≈3.5, S2 має y=5
```

```
- Додаємо S2 до стану
- Стан: [S1, S4, S2]
- S2 сусідить з S4
- Перевіряємо перетин S4-S2
- Знайдено перетин у точці (8/3, 10/3) ≈ (2.67, 3.33)
- Додаємо подію перетину
```

Крок 4: Обробка події x = 8/3 (перетин S4-S2)

```
Подія: перетин S4-S2
- Додаємо точку (8/3, 10/3) до результату
- До перетину: [S1, S4, S2]
- Міняємо порядок S4 та S2
- Після перетину: [S1, S2, S4]
- Нові сусідні пари: S1-S2
- Перевіряємо перетин S1-S2
- Перетин за межами відрізків, не додаємо
```

Крок 5: Обробка події x = 3

```
Подія: початок S3 (3, 3)
- На x=3: S1 має y=3, S2 має y=3, S4 має y=3, S3 має y=3
- Всі проходять через одну точку!
- Додаємо S3, визначаємо порядок за нахилом
- Перевіряємо всі нові сусідні пари

Подія: перетин S1-S4 (3, 3)
- Точка вже оброблена або збігається з початком S3
```

Крок 6-8: Обробка решти подій

Продовжуємо обробляти кінці відрізків, видаляючи їх зі стану та перевіряючи нові сусідні пари.

Фінальний результат:

Знайдені перетини: - (3, 3) - перетин S1 та S4 - (8/3, 10/3) - перетин S2 та S4

Обробка особливих випадків

Вертикальні відрізки:

Вертикальні відрізки потребують спеціальної обробки, оскільки вони перетинають лінію заметання в інтервалі y-координат:

```
Процедура HandleVerticalSegment(segment, status, intersections):
    // Для вертикального відрізка знаходимо всі відрізки,
    // які він перетинає в діапазоні y-координат

    y_min = min(segment.початок.y, segment.кінець.y)
    y_max = max(segment.початок.y, segment.кінець.y)
    x = segment.початок.x
```

```
    // Знаходимо всі відрізки в діапазоні [y_min, y_max]
    segments_in_range = status.знайти_в_діапазоні(y_min, y_max)

    для кожного s в segments_in_range:
        y = GetYAtX(s, x)
        якщо y_min <= y <= y_max:
            intersections.додати(Точка(x, y))
```

Колінеарні відрізки:

Якщо відрізки лежать на одній прямій, вони можуть перетинатися в багатьох точках:

```
Функція HandleCollinearSegments(s1, s2):
    // Визначаємо перекриття колінеарних відрізків
    // Проектуємо на вісь з більшою різницею координат

    якщо abs(s1.кінець.x - s1.початок.x) > abs(s1.кінець.y - s1.початок.y):
        // Проектуємо на вісь X
        overlap_start = max(min(s1.початок.x, s1.кінець.x),
                            min(s2.початок.x, s2.кінець.x))
        overlap_end = min(max(s1.початок.x, s1.кінець.x),
                          max(s2.початок.x, s2.кінець.x))
    інакше:
        // Проектуємо на вісь Y
        // Аналогічно для y-координат

    якщо overlap_start <= overlap_end:
        // Відрізки перекриваються
        // Повертаємо кінці перекриття як точки перетину
```

Аналіз складності

Часова складність:

- Ініціалізація: O(n log n) для сортування подій

- Кожна подія обробляється за O(log n) - операції з деревом

- Максимальна кількість подій: 2n + k, де k - кількість перетинів

- Загальна складність: O((n + k) log n)

Просторова складність:

- Черга подій: O(n + k)

- Стан заметання: O(n)

- Загалом: O(n + k)

Оптимізації алгоритму

1. Використання предикатів для точних обчислень:

```
Функція OrientationPredicate(p, q, r):
    // Точне визначення орієнтації трьох точок
    // Використовує лише цілочисельну арифметику для точок з цілими к
оординатами

    // Обчислюємо визначник матриці
    // | px py 1 |
    // | qx qy 1 |
    // | rx ry 1 |

    det = (qx - px) * (ry - py) - (qy - py) * (rx - px)

    якщо det > 0:
        повернути COUNTERCLOCKWISE
    інакше якщо det < 0:
        повернути CLOCKWISE
    інакше:
        повернути COLLINEAR
```

Область застосування

Алгоритм Бентлі-Оттманна широко використовується в багатьох галузях:

- Комп'ютерна графіка: Відсікання невидимих ліній, побудова карт глибини, обробка полігональних моделей, виявлення самоперетинів контурів

- Географічні інформаційні системи (ГІС): Аналіз топології карт, пошук перехресть доріг, валідація геометричних даних, накладання шарів карт

- САПР (системи автоматизованого проектування): Перевірка коректності креслень, виявлення конфліктів між елементами, оптимізація розміщення компонентів

- Обробка маршрутів: Знаходження перетинів траєкторій, аналіз транспортних потоків, планування повітряних коридорів

- Комп'ютерний зір: Аналіз контурів об'єктів, векторизація растрових зображень, розпізнавання геометричних примітивів

- Робототехніка: Планування шляхів з урахуванням перешкод, аналіз видимості, побудова карт приміщень

Приклади застосування

Приклад 1: Валідація картографічних даних

У ГІС-системах часто потрібно перевірити коректність векторних карт:

```
Алгоритм ValidateMapTopology(road_segments):
    // Знаходимо всі перетини доріг
    intersections = BentleyOttmann(road_segments)

    errors = []
    valid_intersections = []

    для кожного перетину intersection в intersections:
        // Перевіряємо, чи є перетин валідним перехрестям
        segments_at_point = GetSegmentsAtPoint(intersection, road_seg
ments)

        якщо segments_at_point.кількість == 2:
            // Два відрізки - можливо помилка топології
            s1 = segments_at_point[0]
            s2 = segments_at_point[1]

            якщо не IsEndpoint(intersection, s1) або не IsEndpoint(in
tersection, s2):
                // Перетин не в кінцевих точках - помилка
                errors.додати({
                    type: "INVALID_CROSSING",
                    location: intersection,
                    segments: [s1, s2]
                })
            інакше:
                valid_intersections.додати(intersection)
        інакше:
            // Більше двох відрізків - складне перехрестя
            valid_intersections.додати(intersection)

    повернути {
        valid: valid_intersections,
        errors: errors
    }
```

Приклад 2: Оптимізація друкованих плат

При проектуванні друкованих плат потрібно мінімізувати перетини провідників:

```
Алгоритм OptimizePCBLayout(traces, components):
    // traces - провідники на платі
    // components - розміщені компоненти
```

```
    // Знаходимо всі перетини провідників
    crossings = BentleyOttmann(traces)

    // Групуємо перетини за шарами
    crossings_by_layer = GroupByLayer(crossings)

    optimization_suggestions = []

    для кожного layer, layer_crossings в crossings_by_layer:
        якщо layer_crossings.кількість > THRESHOLD:
            // Занадто багато перетинів на одному шарі

            // Знаходимо провідники з найбільшою кількістю перетинів
            problematic_traces = FindMostCrossingTraces(layer_crossin
gs, traces)

            для кожного trace в problematic_traces:
                // Пропонуємо альтернативні маршрути
                alternative = FindAlternativeRoute(trace, components)

                якщо alternative != null:
                    new_crossings = CountCrossings(alternative, trace
s)

                    якщо new_crossings < GetCrossingCount(trace, cros
sings):
                        optimization_suggestions.додати({
                            original: trace,
                            alternative: alternative,
                            improvement: GetCrossingCount(trace) - ne
w_crossings
                        })

    повернути optimization_suggestions
```

Приклад 3: Аналіз повітряного простору

Для управління повітряним рухом важливо виявляти перетини траєкторій:

```
Алгоритм AirspaceConflictDetection(flight_paths, time_window):
    // Проектуємо 4D траєкторії (x, y, z, t) на 2D площину (x, y)
    // для кожного часового зрізу

    conflicts = []
    time_slices = GenerateTimeSlices(time_window, SLICE_DURATION)

    для кожного time_slice в time_slices:
```

```
        // Отримуємо проекції траєкторій на цей момент часу
        segments_2d = []

        для кожного path в flight_paths:
            segment = GetTrajectorySegmentAtTime(path, time_slice)
            якщо segment != null:
                segments_2d.додати({
                    segment: Project2D(segment),
                    flight: path.flight_id,
                    altitude: segment.altitude
                })

        // Знаходимо перетини в 2D
        intersections_2d = BentleyOttmann(segments_2d.map(s => s.segm
ent))

        // Перевіряємо висоту для кожного перетину
        для кожного intersection в intersections_2d:
            flights = GetFlightsAtIntersection(intersection, segments
_2d)

            якщо flights.кількість >= 2:
                alt_diff = abs(flights[0].altitude - flights[1].altit
ude)

                якщо alt_diff < MIN_VERTICAL_SEPARATION:
                    conflicts.додати({
                        time: time_slice,
                        location: intersection,
                        flights: flights,
                        severity: CalculateSeverity(alt_diff)
                    })

    повернути conflicts
```

Переваги та недоліки

Переваги:

- Оптимальна складність для виведення: O((n + k) log n) є оптимальною для задачі з k перетинами

- Ефективність для розріджених перетинів: Коли $k << n^2$, алгоритм значно швидший за наївний $O(n^2)$

- Елегантність рішення: Використання техніки заметання площини робить алгоритм концептуально простим

- Можливість ранньої зупинки: Можна зупинити алгоритм після знаходження першого перетину

- Розширюваність: Легко модифікується для підрахунку перетинів, знаходження найлівішого перетину тощо

- Теоретична база: Добре вивчений алгоритм з доведеною коректністю

Недоліки:

- Складність реалізації: Потребує акуратної обробки багатьох граничних випадків

- Числова нестабільність: Чутливий до помилок округлення при роботі з дійсними числами

- Погана поведінка для щільних перетинів: При $k \approx n^2$ складність наближається до $O(n^2)$

- Складні структури даних: Потребує ефективної реалізації збалансованого дерева

- Обробка виродженостей: Вертикальні відрізки, співпадаючі точки потребують спеціальної обробки

- Використання пам'яті: $O(n + k)$ пам'яті може бути значним для великих k

Контрольні запитання

1. Поясніть основну ідею техніки заметання площини в алгоритмі Бентлі-Оттманна.

2. Які типи подій обробляє алгоритм і чому саме ці три типи?

3. Чому важливо перевіряти перетини лише між сусідніми відрізками в стані заметання?

4. Як змінюється порядок відрізків у стані заметання при проходженні через точку перетину?

5. Яка роль пріоритетної черги в алгоритмі?

6. Як обробляються вертикальні відрізки?

7. Чому складність алгоритму залежить від кількості перетинів k?

8. Які проблеми виникають при роботі з числами з плаваючою точкою?

9. Як модифікувати алгоритм для підрахунку кількості перетинів без їх явного знаходження?

10. Які альтернативні підходи існують для розв’язання задачі пошуку перетинів?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Бентлі-Оттманна для пошуку перетинів відрізків

2. Мета

Вивчити техніку заметання площини, навчитися працювати зі складними геометричними алгоритмами, освоїти використання збалансованих дерев для динамічного впорядкування

3. Завдання

- Реалізувати структури даних для подій та стану заметання
- Створити функції обчислення перетинів відрізків
- Реалізувати основний алгоритм з обробкою всіх типів подій
- Додати візуалізацію процесу заметання
- Обробити особливі випадки (вертикальні, колінеарні відрізки)
- Порівняти з наївним алгоритмом
- Дослідити поведінку для різних типів даних

4. Опис

Лабораторна робота передбачає повну реалізацію алгоритму Бентлі-Оттманна з візуалізацією руху лінії заметання та динамічної зміни стану. Особлива увага приділяється коректній обробці граничних випадків та числовій стабільності.

5. Порядок виконання

- Створити класи для точок, відрізків та подій
- Реалізувати пріоритетну чергу для подій
- Створити збалансоване дерево для стану заметання
- Реалізувати функцію обчислення перетину двох відрізків
- Написати обробники для кожного типу подій
- Створити функцію порівняння відрізків для дерева стану
- Реалізувати основний цикл алгоритму
- Додати візуалізацію з покроковим виконанням

- Створити генератори тестових даних

- Провести тестування та оптимізацію

6. Результат

Програма з графічним інтерфейсом, яка знаходить всі перетини у множині відрізків, візуалізує процес роботи алгоритму, дозволяє покрокове виконання та аналіз продуктивності.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:

- Клас для представлення подій з правильним порядком сортування

- Реалізація пріоритетної черги

- Реалізація збалансованого дерева для стану заметання

- Функція точного обчислення перетину відрізків

- Обробники всіх типів подій

- Функція порівняння відрізків у стані

- Модуль візуалізації

- Обробка особливих випадків

- Продемонструвати роботу додатку:

- Візуалізація руху лінії заметання

- Відображення поточного стану (відрізки в дереві)

- Підсвічування знайдених перетинів

- Покрокове виконання з поясненнями

- Робота з різними наборами відрізків

- Обробка вертикальних відрізків

- Порівняння часу роботи з наївним алгоритмом

Бути готовим відповісти на питання:

- Чому перевіряються лише сусідні відрізки?

- Як довести коректність алгоритму?

- Які існують альтернативи збалансованому дереву?

- Як забезпечити числову стабільність?

- Коли алгоритм працює найгірше?

- Як розширити для кривих замість відрізків?

Тести

1. Яка часова складність алгоритму Бентлі-Оттманна?

1) $O(n^2)$

2) $O(n \log n)$

3) $O((n + k) \log n)$

4) $O(nk)$

2. Які події обробляє алгоритм?

1) Лише початки відрізків

2) Початки та кінці відрізків

3) Початки, кінці та перетини

4) Лише перетини

3. Що зберігається в стані заметання?

1) Всі відрізки

2) Відрізки, що перетинають поточну лінію заметання

3) Знайдені перетини

4) Події

4. В якому порядку впорядковані відрізки в стані заметання?

1) За x-координатою

2) За довжиною

3) За y-координатою на лінії заметання

4) За часом додавання

5. Коли два відрізки можуть мати перетин?

1) Завжди

2) Коли вони сусідні в стані заметання

3) Коли вони паралельні

4) Коли вони вертикальні

Відповіді:

1. 3)

2. 3)

3. 2)

4. 3)

5. 2)

Тести підвищеної складності

1. Що відбувається з порядком двох відрізків у стані після їх перетину?

1) Порядок не змінюється

2) Вони міняються місцями

3) Обидва видаляються

4) Додається новий відрізок

2. Скільки максимально подій може бути в черзі для n відрізків?

1) n

2) 2n

3) 2n + k

4) n^2

3. Яка структура даних найкраще підходить для стану заметання?

1) Масив

2) Хеш-таблиця

3) Збалансоване бінарне дерево

4) Черга

4. Як обробляються вертикальні відрізки?

1) Ігноруються

2) Як звичайні відрізки

3) Потребують спеціальної обробки

4) Розбиваються на горизонтальні

5. Коли алгоритм має найгіршу складність?

1) Коли всі відрізки паралельні

2) Коли $k \approx n^2$

3) Коли всі відрізки вертикальні

4) Коли немає перетинів

6. Що робити, якщо три відрізки перетинаються в одній точці?

1) Обробити як три окремі перетини

2) Обробити як одну подію

3) Ігнорувати

4) Викинути помилку

7. Як впливає точність обчислень на роботу алгоритму?

1) Не впливає

2) Може призвести до пропуску перетинів

3) Лише сповільнює роботу

4) Покращує результат

8. Чи можна модифікувати алгоритм для роботи з променями?

1) Ні, потрібен інший алгоритм

2) Так, без змін

3) Так, з незначними модифікаціями

4) Лише для горизонтальних променів

9. Яка операція найдорожча в алгоритмі?

1) Обчислення перетину

2) Операції з деревом стану

3) Сортування подій

4) Візуалізація

10. Як знайти лише кількість перетинів без їх координат?

1) Неможливо з цим алгоритмом

2) Лічити події перетину без обчислення координат

3) Використати інший алгоритм

4) Модифікувати обчислення перетину

Відповіді:

1. 2) Вони міняються місцями в порядку стану заметання

2. 3) 2n початкових подій плюс k подій перетину

3. 3) Збалансоване дерево забезпечує O(log n) для всіх операцій

4. 3) Вертикальні відрізки перетинають діапазон y-координат

5. 2) Коли кількість перетинів близька до максимальної

6. 1) Кожна пара обробляється окремо

7. 2) Помилки округлення можуть призвести до неправильного порядку подій

8. 3) Промені - це відрізки з одним нескінченним кінцем

9. 2) Операції вставки/видалення/пошуку в дереві виконуються найчастіше

10. 2) Можна лічити події без обчислення точних координат

Завдання для самостійної роботи

1. Реалізуйте версію алгоритму з використанням точної арифметики для уникнення числових помилок.

2. Розширте алгоритм для роботи з відрізками різних кольорів, знаходячи лише перетини відрізків різних кольорів.

3. Створіть паралельну версію алгоритму, розділивши площину на вертикальні смуги.

4. Реалізуйте модифікацію для знаходження всіх k-кратних перетинів (точок, де перетинаються k відрізків).

5. Розробіть версію для роботи з дугами кіл замість прямих відрізків.

6. Створіть алгоритм для динамічного набору відрізків з додаванням та видаленням.

7. Реалізуйте знаходження найлівішого перетину без обчислення всіх.

8. Розробіть версію з апроксимацією для швидкого знаходження “майже перетинів”.

9. Створіть модифікацію для відрізків з вагами, де потрібні лише перетини з сумарною вагою більше порогу.

10. Реалізуйте алгоритм для тривимірного простору (перетин трикутників).

Теми лабораторних робіт

1. Розробка системи валідації картографічних даних з виявленням топологічних помилок.

2. Створення редактора векторної графіки з автоматичним виявленням самоперетинів.

3. Реалізація системи аналізу транспортних потоків з визначенням проблемних перехресть.

4. Розробка інструменту оптимізації друкованих плат з мінімізацією перетинів.

5. Створення системи планування повітряних маршрутів з уникненням конфліктів.

6. Реалізація алгоритму векторизації растрових зображень з обробкою перетинів.

7. Розробка геометричного вирішувача для САПР з перевіркою обмежень.

8. Створення системи аналізу міських мереж (водопровід, електрика) з пошуком перетинів.

9. Реалізація інструменту для обробки лазерного різання з оптимізацією траєкторій.

10. Розробка системи моделювання руху роботів з виявленням колізій траєкторій.

11.3.2. Алгоритм променевого трасування

Теоретичні відомості

Основні поняття та визначення

Променеве трасування (Ray Tracing) — це техніка рендерингу в комп'ютерній графіці, яка моделює фізичну поведінку світла для створення реалістичних зображень. Алгоритм базується на принципі зворотного відстеження променів світла від камери (ока спостерігача) через кожен піксель екрану до об'єктів сцени.

Математична основа алгоритму

Промінь у тривимірному просторі описується параметричним рівнянням:

$$P(t) = O + t \cdot D$$

де:

- $P(t)$ — точка на промені в момент часу t

- O — початкова точка променя

- D — нормалізований вектор напрямку променя

- t — параметр, що визначає відстань від початкової точки ($t \geq 0$)

Основні компоненти алгоритму

- Камера (Camera) — визначає точку спостереження та площину проекції

- Промінь (Ray) — математичний об'єкт, що описує шлях світла

- Об'єкти сцени (Scene Objects) — геометричні примітиви (сфери, площини, трикутники)

- Джерела світла (Light Sources) — точкові, напрямлені або площинні джерела

- Матеріали (Materials) — властивості поверхонь об’єктів

Алгоритм трасування первинних променів такий:

```
FUNCTION RayTrace(scene, camera, width, height):
    image = CreateImage(width, height)

    FOR y = 0 TO height-1:
        FOR x = 0 TO width-1:
            // Обчислення напрямку променя через піксель
            ray = GenerateRay(camera, x, y, width, height)

            // Трасування променя
            color = TraceRay(ray, scene, 0)

            // Запис кольору в зображення
            image[x][y] = color
        END FOR
    END FOR

    RETURN image
END FUNCTION
```

Генерація променя через піксель

```
FUNCTION GenerateRay(camera, x, y, width, height):
    // Перетворення координат пікселя в координати площини проекції
    u = (x + 0.5) / width
    v = (y + 0.5) / height

    // Обчислення точки на площині проекції
    aspectRatio = width / height
    fov = camera.fieldOfView

    // Переведення в систему координат камери
    px = (2 * u - 1) * tan(fov/2) * aspectRatio
    py = (1 - 2 * v) * tan(fov/2)

    // Створення променя
    origin = camera.position
    direction = Normalize(camera.forward + px * camera.right + py * c
amera.up)

    RETURN Ray(origin, direction)
END FUNCTION
```

Наведемо алгоритм пошуку перетинів з об’єктами

```
FUNCTION FindIntersection(ray, objects):
    closestT = INFINITY
    closestObject = NULL
    hitPoint = NULL
    normal = NULL

    FOR EACH object IN objects:
        t = IntersectRayObject(ray, object)

        IF t > 0 AND t < closestT:
            closestT = t
            closestObject = object
            hitPoint = ray.origin + t * ray.direction
            normal = CalculateNormal(object, hitPoint)
        END IF
    END FOR

    RETURN (closestObject, hitPoint, normal, closestT)
END FUNCTION
```

Перетин променя зі сферою

Для сфери з центром C і радіусом r, перетин знаходиться розв’язанням квадратного рівняння:

```
FUNCTION IntersectRaySphere(ray, sphere):
    // Вектор від початку променя до центру сфери
    oc = ray.origin - sphere.center

    // Коефіцієнти квадратного рівняння at² + bt + c = 0
    a = DotProduct(ray.direction, ray.direction)
    b = 2.0 * DotProduct(oc, ray.direction)
    c = DotProduct(oc, oc) - sphere.radius * sphere.radius

    // Дискримінант
    discriminant = b*b - 4*a*c

    IF discriminant < 0:
        RETURN NO_INTERSECTION
    END IF

    // Два корені
    t1 = (-b - sqrt(discriminant)) / (2*a)
    t2 = (-b + sqrt(discriminant)) / (2*a)
```

```
    // Повертаємо найближчий позитивний корінь
    IF t1 > 0:
        RETURN t1
    ELSE IF t2 > 0:
        RETURN t2
    ELSE:
        RETURN NO_INTERSECTION
    END IF
END FUNCTION
```

Модель освітлення Фонга

Модель освітлення Фонга обчислює колір точки як суму трьох компонентів:

I = I_ambient + I_diffuse + I_specular

де: - I_ambient — фонове освітлення - I_diffuse — дифузне відбиття - I_specular — дзеркальне відбиття

```
FUNCTION CalculateLighting(point, normal, viewDir, material, lights):
    color = material.ambient * ambientLight

    FOR EACH light IN lights:
        // Вектор до джерела світла
        lightDir = Normalize(light.position - point)

        // Перевірка тіні
        shadowRay = Ray(point + EPSILON * normal, lightDir)
        IF IsInShadow(shadowRay, light):
            CONTINUE
        END IF

        // Дифузна компонента
        diffuse = max(0, DotProduct(normal, lightDir))
        color += material.diffuse * light.color * diffuse

        // Дзеркальна компонента
        reflectDir = Reflect(-lightDir, normal)
        specular = pow(max(0, DotProduct(reflectDir, viewDir)), mater
ial.shininess)
        color += material.specular * light.color * specular
    END FOR

    RETURN color
END FUNCTION
```

Рекурсивне трасування променів

```
FUNCTION TraceRay(ray, scene, depth):
    IF depth > MAX_DEPTH:
        RETURN BACKGROUND_COLOR
    END IF

    // Знаходимо найближчий перетин
    (object, hitPoint, normal, t) = FindIntersection(ray, scene.objec
ts)

    IF object == NULL:
        RETURN BACKGROUND_COLOR
    END IF

    // Обчислюємо локальне освітлення
    viewDir = -ray.direction
    color = CalculateLighting(hitPoint, normal, viewDir, object.mater
ial, scene.lights)

    // Відбитий промінь
    IF object.material.reflectivity > 0:
        reflectDir = Reflect(ray.direction, normal)
        reflectRay = Ray(hitPoint + EPSILON * normal, reflectDir)
        reflectColor = TraceRay(reflectRay, scene, depth + 1)
        color += object.material.reflectivity * reflectColor
    END IF

    // Заломлений промінь
    IF object.material.transparency > 0:
        refractDir = Refract(ray.direction, normal, object.material.r
efractiveIndex)
        IF refractDir != NULL:
            refractRay = Ray(hitPoint - EPSILON * normal, refractDir)
            refractColor = TraceRay(refractRay, scene, depth + 1)
            color += object.material.transparency * refractColor
        END IF
    END IF

    RETURN color
END FUNCTION
```

Алгоритм обчислення тіней

```
FUNCTION IsInShadow(shadowRay, light):
    maxDistance = Distance(shadowRay.origin, light.position)

    FOR EACH object IN scene.objects:
```

```
        t = IntersectRayObject(shadowRay, object)
        IF t > 0 AND t < maxDistance:
            RETURN TRUE
        END IF
    END FOR

    RETURN FALSE
END FUNCTION
```

Антиаліасинг методом суперсемплінгу

```
FUNCTION RayTraceWithAntialiasing(scene, camera, width, height, sampl
esPerPixel):
    image = CreateImage(width, height)

    FOR y = 0 TO height-1:
        FOR x = 0 TO width-1:
            color = (0, 0, 0)

            // Випадкові зміщення в межах пікселя
            FOR sample = 0 TO samplesPerPixel-1:
                dx = Random() - 0.5
                dy = Random() - 0.5

                ray = GenerateRay(camera, x + dx, y + dy, width, heig
ht)
                color += TraceRay(ray, scene, 0)
            END FOR

            // Усереднення кольору
            image[x][y] = color / samplesPerPixel
        END FOR
    END FOR

    RETURN image
END FUNCTION
```

Область застосування

Алгоритм променевого трасування широко використовується в різних галузях комп’ютерної графіки та візуалізації:

- Кіноіндустрія та анімація — створення фотореалістичних спецефектів, рендеринг складних сцен з реалістичним освітленням, відбиттями та тінями. Студії Pixar, DreamWorks, Industrial Light & Magic використовують ray tracing для фінального рендерингу

- Архітектурна візуалізація — створення реалістичних зображень будівель та інтер'єрів для презентацій проектів, аналіз природного освітлення приміщень

- Дизайн продуктів — візуалізація промислових виробів з точним відтворенням матеріалів (метал, скло, пластик), рендеринг ювелірних виробів

- Наукова візуалізація — моделювання оптичних систем (лінзи, телескопи, мікроскопи), візуалізація медичних даних (МРТ, КТ)

- Ігрова індустрія — з появою апаратного прискорення (NVIDIA RTX, AMD RDNA 2) ray tracing використовується для рендерингу в реальному часі

- Віртуальна та доповнена реальність — створення реалістичних віртуальних середовищ з правильним освітленням

Приклади застосування

Приклад 1: Рендеринг простої сцени з трьома сферами

Розглянемо покроковий процес рендерингу сцени з трьома сферами різних матеріалів:

```
FUNCTION RenderSimpleScene():
    // Визначення сцени
    sphere1 = Sphere(center=(0, 0, -5), radius=1, material=Metal(color=red, fuzz=0.3))
    sphere2 = Sphere(center=(2, 0, -5), radius=1, material=Glass(refractiveIndex=1.5))
    sphere3 = Sphere(center=(-2, 0, -5), radius=1, material=Diffuse(color=blue))

    light = PointLight(position=(0, 5, -3), color=white, intensity=100)

    camera = Camera(
        position=(0, 0, 0),
        lookAt=(0, 0, -5),
        up=(0, 1, 0),
        fov=60,
        aspectRatio=16/9
    )

    // Рендеринг кожного пікселя
    FOR y = 0 TO imageHeight-1:
        FOR x = 0 TO imageWidth-1:
```

```
            ray = camera.GetRay(x, y)
            color = TraceRay(ray, [sphere1, sphere2, sphere3], [light
], 0)
            SetPixel(x, y, color)
        END FOR
    END FOR
END FUNCTION
```
Для кожного пікселя виконуються наступні кроки:

- Генерація променя від камери через піксель

- Перевірка перетину з усіма об’єктами сцени

- Для найближчого перетину:

- Обчислення нормалі в точці перетину

- Трасування тіньового променя до джерела світла

- Обчислення дифузної та дзеркальної компонент освітлення

- Рекурсивне трасування відбитих/заломлених променів

Приклад 2: Моделювання каустики

Каустика — це світлові візерунки, що утворюються при фокусуванні світла криволінійними поверхнями:

```
FUNCTION SimulateCaustics():
    // Склянка води на столі
    table = Plane(y=-2, material=Diffuse(color=wood_texture))
    glass = Cylinder(center=(0, 0, 0), radius=1, height=3,
                     material=Glass(refractiveIndex=1.33))
    water = Cylinder(center=(0, 0.1, 0), radius=0.95, height=2.8,
                     material=Water(refractiveIndex=1.33))

    // Направлене світло зверху
    sunlight = DirectionalLight(direction=(0, -1, -0.2), color=white)

    // Фотонне картування для каустики
    photonMap = EmitPhotons(sunlight, numPhotons=1000000)

    FOR EACH photon IN photonMap:
        ray = Ray(photon.origin, photon.direction)

        // Трасування через скло та воду
        IF Intersect(ray, glass):
            refractedRay = Refract(ray, glass.normal, 1.0/1.5)
            IF Intersect(refractedRay, water):
                finalRay = Refract(refractedRay, water.normal, 1.5/1.
```

```
33)
                IF Intersect(finalRay, table):
                    StorePhoton(finalRay.hitPoint, photon.energy)
                END IF
            END IF
        END IF
    END FOR

    // Візуалізація фотонної карти
    RenderPhotonMap(photonMap)
END FUNCTION
```

Приклад 3: Глобальне освітлення методом трасування шляхів

```
FUNCTION PathTracing(ray, scene, depth):
    IF depth > MAX_DEPTH:
        RETURN BLACK
    END IF

    hit = FindIntersection(ray, scene)
    IF hit == NULL:
        RETURN EnvironmentLight(ray.direction)
    END IF

    // Прямє освітлення
    directLight = BLACK
    FOR EACH light IN scene.lights:
        shadowRay = Ray(hit.point, DirectionTo(light))
        IF NOT InShadow(shadowRay, scene):
            directLight += BRDF(hit.material, ray.direction, shadowRa
y.direction)
                           * light.intensity * cos(angle)
        END IF
    END FOR

    // Непряме освітлення (рекурсивно)
    indirectDirection = SampleHemisphere(hit.normal)
    indirectRay = Ray(hit.point, indirectDirection)
    indirectLight = PathTracing(indirectRay, scene, depth + 1)

    // Комбінування результатів
    RETURN hit.material.emission + directLight +
           BRDF(hit.material, ray.direction, indirectDirection) * ind
irectLight
END FUNCTION
```

Переваги та недоліки

Переваги алгоритму променевого трасування:

- Фізична коректність — алгоритм базується на законах геометричної оптики, що дозволяє отримувати реалістичні зображення

- Природні ефекти — тіні, відбиття, заломлення світла виникають автоматично без додаткових алгоритмів

- Простота реалізації базового алгоритму — основний алгоритм зрозумілий та відносно простий у реалізації

- Масштабованість — легко додавати нові типи об'єктів та матеріалів без зміни основного алгоритму

- Підтримка складної геометрії — можливість рендерингу процедурних поверхонь, CSG-об'єктів

- Глобальне освітлення — можливість моделювання непрямого освітлення, каустики, кольорового відбиття

- Паралелізація — кожен піксель обчислюється незалежно, що дозволяє ефективно розпаралелювати обчислення

Недоліки алгоритму променевого трасування:

- Висока обчислювальна складність — O(N×M×K), де N — кількість пікселів, M — кількість об'єктів, K — глибина рекурсії

- Повільна швидкість — для якісного рендерингу потрібні години обчислень навіть на потужному обладнанні

- Складність оптимізації — необхідність використання просторових структур даних (BVH, kd-tree, octree)

- Проблема шуму — при моделюванні глобального освітлення виникає шум, що вимагає великої кількості семплів

- Обмеження в реальному часі — навіть з апаратним прискоренням складно досягти високої частоти кадрів для складних сцен

- Велике споживання пам'яті — для зберігання просторових структур та текстур високої роздільності

- Складність моделювання деяких ефектів — розсіювання в об'ємі (туман, дим), підповерхневе розсіювання

Контрольні запитання

1. У чому полягає основна ідея алгоритму променевого трасування? Чим він відрізняється від растеризації?

2. Опишіть математичну модель променя та основні параметри, що його визначають.

3. Як обчислюється перетин променя зі сферою? Наведіть формули та поясніть фізичний зміст.

4. Що таке модель освітлення Фонга? З яких компонентів вона складається?

5. Поясніть принцип рекурсивного трасування променів. Навіщо потрібно обмежувати глибину рекурсії?

6. Як реалізується моделювання тіней в алгоритмі променевого трасування?

7. Що таке закон Снелліуса і як він використовується для моделювання заломлення світла?

8. Які методи антиаліасингу можна застосувати в променевому трасуванні?

9. Поясніть призначення просторових структур даних (BVH, kd-tree) в контексті оптимізації.

10. Які основні відмінності між класичним променевим трасуванням та трасуванням шляхів?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація базового алгоритму променевого трасування для візуалізації тривимірних сцен

2. Мета

Вивчити принципи роботи алгоритму променевого трасування, реалізувати основні компоненти рендерера, дослідити вплив параметрів на якість та швидкість рендерингу.

3. Завдання

- Реалізувати генерацію променів через пікселі екрану

- Реалізувати перетин променя з примітивами (сфера, площина)

- Реалізувати модель освітлення Фонга

- Додати підтримку тіней

- Реалізувати рекурсивне трасування для відбиттів

- Додати антиаліасинг

- Провести аналіз продуктивності

4. Опис

Лабораторна робота передбачає поетапну реалізацію компонентів ray tracer'а. Починаючи з простого рендерингу сфер з базовим освітленням, студенти поступово додають складніші функції: тіні, відбиття, заломлення, різні типи матеріалів. Фінальна програма повинна рендерити сцену з декількома об'єктами, демонструючи всі реалізовані ефекти.

5. Порядок виконання

- Створити базову структуру проекту з класами Vector3, Ray, Camera
- Реалізувати клас Sphere з методом перетину променя
- Створити простий рендерер, що малює сфери одним кольором
- Додати клас Material та реалізувати дифузне освітлення
- Реалізувати трасування тіньових променів
- Додати дзеркальні відбиття (рекурсивне трасування)
- Реалізувати клас Plane для створення "підлоги"
- Додати суперсемплінг для антиаліасингу
- Оптимізувати код використовуючи багатопоточність
- Створити демонстраційну сцену з різними матеріалами

6. Результат

Програма, що рендерить тривимірну сцену методом променевого трасування з підтримкою базових оптичних ефектів. Зображення повинно демонструвати коректні тіні, відбиття, різні типи матеріалів. Додатково: графіки залежності часу рендерингу від роздільності та кількості об'єктів.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас Ray з методами генерації та трасування
- Класи геометричних примітивів (Sphere, Plane) з методами перетину
- Клас Material з реалізацією BRDF
- Клас Camera з методом генерації променів
- Основний цикл рендерингу з підтримкою багатопоточності

2. Продемонструвати роботу додатку:

- Рендеринг сцени з мінімум 5 об'єктами
- Демонстрація роботи тіней від точкового джерела світла

- Показ дзеркальних відбиттів з різною глибиною рекурсії
- Порівняння зображень з антиаліасингом та без нього
- Вимірювання часу рендерингу для різних налаштувань

3. Бути готовим відповісти на питання:
- Як визначається видимість точки з джерела світла?
- Чому виникають артефакти “acne” і як їх уникнути?
- Як впливає кількість семплів на якість антиаліасингу?
- Які оптимізації можна застосувати для прискорення рендерингу?
- Як реалізувати м’які тіні від площинного джерела світла?

Тести

1. Основна ідея променевого трасування полягає в:

1) Проекції вершин на площину екрану

2) Трасуванні променів від камери через пікселі до об’єктів сцени

3) Растеризації трикутників по рядках

4) Обчисленні освітлення у вершинах

2. Параметричне рівняння променя має вигляд:

1) $P = O \times t + D$

2) $P(t) = O + t \cdot D$

3) $P = O - t \cdot D$

4) $P(t) = D + t \cdot O$

3. Для перевірки видимості точки з джерела світла використовується:

1) Первинний промінь

2) Відбитий промінь

3) Тіньовий промінь

4) Заломлений промінь

4. Модель освітлення Фонга НЕ включає компоненту:

1) Фонову (ambient)

2) Дифузну (diffuse)

3) Дзеркальну (specular)

4) Глобальну (global)

5. Антиаліасинг методом суперсемплінгу передбачає:

1) Розмиття готового зображення

2) Трасування декількох променів через кожен піксель

3) Зменшення роздільності зображення

4) Використання текстурної фільтрації

Відповіді:

1-2,

2-3,

3-3,

4-4,

5-2

Тести підвищеної складності

1. При обчисленні перетину променя зі сферою дискримінант квадратного рівняння від’ємний. Це означає:

1) Промінь перетинає сферу в одній точці

2) Промінь не перетинає сферу

3) Промінь дотикається до сфери

4) Промінь проходить через центр сфери

2. Яка структура даних найефективніша для прискорення пошуку перетинів у сценах з великою кількістю об’єктів?

1) Лінійний масив

2) Хеш-таблиця

3) BVH

4) Черга з пріоритетами

3. Закон Снелліуса для заломлення світла виражається формулою:

1) $n_1 \cdot cos(\theta_1) = n_2 \cdot cos(\theta_2)$

2) $n_1 \cdot sin(\theta_1) = n_2 \cdot sin(\theta_2)$

3) $n_1 \cdot tan(\theta_1) = n_2 \cdot tan(\theta_2)$

4) $n_1/sin(\theta_1) = n_2/sin(\theta_2)$

4. Явище повного внутрішнього відбиття виникає коли:

1) Кут падіння менший за критичний кут

2) Промінь переходить з менш щільного середовища в більш щільне

3) Промінь переходить з більш щільного середовища в менш щільне і кут падіння перевищує критичний

4) Показники заломлення середовищ однакові

5. Метод “Вибірка за важливістю” в контексті променевого трасування використовується для:

1) Вибору найважливіших об’єктів для рендерингу

2) Генерації променів з розподілом, що відповідає BRDF

3) Сортування пікселів за важливістю

4) Видалення невидимих об’єктів

6. Руська рулетка застосовується при трасуванні для:

1) Випадкового вибору матеріалів

2) Обмеження глибини рекурсії ймовірнісним методом

3) Генерації випадкових сцен

4) Розподілу променів між процесорами

7. Які координати використовуються для семплювання напрямків на півсфері?

1) Декартові (x, y, z)

2) Сферичні (θ, φ)

3) Циліндричні (r, φ, z)

4) Барицентричні

8. Для коректного обчислення нормалі в точці перетину променя з трикутником використовується:

1) Крос-добуток ребер трикутника

2) Барицентрична інтерполяція нормалей у вершинах

3) Градієнт неявної функції

4) Усереднення нормалей сусідніх граней

9. Алгоритм трасування призначений для:

1) Генерації випадкових чисел

2) Ефективного обчислення перетину променя з трикутником

3) Побудови BVH-дерева

4) Обчислення освітлення

10. Двонаправлене трасування відрізняється від звичайного тим, що:

1) Використовує два процесори одночасно

2) Трасує промені в обох напрямках

3) Трасує шляхи від камери та від джерел світла з подальшим з'єднанням

4) Обчислює прямє та відбите освітлення окремо

Відповіді:

1-2. Від'ємний дискримінант означає відсутність дійсних коренів рівняння, тобто промінь не перетинає сферу.

2-3. BVH дозволяє ефективно відсікати групи об'єктів, зменшуючи кількість перевірок перетинів з $O(n)$ до $O(log\ n)$.

3-2. Закон Снелліуса пов'язує показники заломлення та синуси кутів падіння і заломлення.

4-3. Повне внутрішнє відбиття виникає при переході в оптично менш щільне середовище, коли $sin(\theta_2) > 1$.

5-2. Вибірка за важливістю генерує напрямки з вищою ймовірністю там, де BRDF має більші значення, зменшуючи дисперсію.

6-2. Руська рулетка дозволяє обмежити глибину трасування статистично коректним способом без систематичної похибки.

7-2. Сферичні координати природно параметризують напрямки на сфері через два кути.

8-2. Для зглажене зітнення використовується інтерполяція нормалей у вершинах за барицентричними координатами.

9-2. Алгоритм використовує барицентричні координати для ефективного обчислення перетину без попереднього обчислення площини.

10-3. BDPT будує шляхи з обох кінців і з'єднує їх, що ефективніше для певних типів освітлення.

Завдання для самостійної роботи

1. Реалізуйте підтримку текстур для об'єктів сцени. Додайте UV-відображення для сфери та площини, реалізуйте білінійну фільтрацію текстур.

2. Розробіть алгоритм для рендерингу м’яких тіней від площинних джерел світла. Використайте метод розподіленого трасування променів.

3. Додайте підтримку об’ємного розсіювання для моделювання туману або диму. Реалізуйте променеве трасування через об’єм.

4. Реалізуйте процедурні текстури (шум Перліна, мармур, дерево) та bump mapping для створення рельєфних поверхонь без зміни геометрії.

5. Створіть адаптивний антиаліасинг, що автоматично збільшує кількість семплів у областях з високим контрастом.

6. Реалізуйте підтримку конструктивної блокової геометрії (CSG) з операціями об’єднання, перетину та віднімання об’єктів.

7. Додайте фізично коректні матеріали на основі мікрограневої теорії з моделями GGX або Beckmann.

8. Оптимізуйте рендерер використовуючи SIMD-інструкції (SSE/AVX) для векторних обчислень. Порівняйте продуктивність.

9. Реалізуйте метод фотонних карт для ефективного моделювання каустики та непрямого освітлення.

10. Створіть інтерактивний режим попереднього перегляду з прогресивним покращенням якості використовуючи метод прогресивного покращення.

Теми лабораторних робіт

1. Базова реалізація променевого трасування з підтримкою сфер та дифузного освітлення.

2. Розширення рендерера: додавання різних геометричних примітивів (площина, куб, циліндр, конус).

3. Реалізація складних матеріалів: метал з різним ступенем шорсткості, діелектрики зі заломленням, анізотропні матеріали.

4. Побудова та використання просторових структур даних (BVH, kd-tree) для прискорення рендерингу.

5. Реалізація різних моделей камери: перспективна, ортографічна, «рибяче око», панорамна 360°.

6. Створення системи анімації з інтерполяцією положень об’єктів та камери, рендеринг відеопослідовності.

7. Розробка плагінної архітектури для розширення функціональності рендерера новими типами об’єктів та матеріалів.

8. Реалізація GPU-версії променевого трасування з використанням CUDA або OpenCL.

9. Інтеграція рендерера з форматами обміну 3D-даними (OBJ, FBX) для імпорту складних моделей.

10. Розробка системи розподілених обчислень для рендерингу на кластері комп’ютерів з балансуванням навантаження.

11.4. Геометричні структури даних: ефективна організація простору

Коли ми працюємо з великими обсягами просторових даних, простих масивів чи списків стає недостатньо. Уявіть собі завдання рендерингу сцени з мільйонами полігонів, пошук найближчого ресторану на карті міста або визначення зіткнень об'єктів у фізичному симуляторі. Наївний підхід, що передбачає перевірку кожного об'єкта, має складність $O(n)$ і стає неприйнятно повільним при збільшенні кількості даних.

Геометричні структури розроблені для того, щоб ефективно організовувати та індексувати об’єкти (точки, лінії, багатокутники) у дво- або тривимірному просторі, дозволяючи виконувати складні запити значно швидше, ніж повний перебір.

Вони дозволяють значно прискорити типові запити, такі як:

1. Пошук найближчого сусіда — знайти об'єкт, найближчий до заданої точки
2. Пошук у діапазоні — знайти всі об'єкти в заданій області простору
3. Перетин променя з об'єктами — визначити, які об'єкти перетинає промінь
4. Виявлення зіткнень — знайти пари об'єктів, що перетинаються

У цьому розділі ми розглянемо три ключові деревоподібні структури:

- **KD-дерево (k-dimensional tree):** Це узагальнення бінарного дерева пошуку для роботи з k вимірами. На кожному рівні дерево ділить простір на дві половини за допомогою площини, що проходить через одну з осей координат. Це робить його ідеальним для швидкого пошуку найближчих сусідів та діапазонних запитів по точкових даних.

- **R-дерево (Rectangle tree):** На відміну від KD-дерева, яке ділить сам простір, R-дерево групує сусідні об'єкти у мінімальні обмежуючі прямокутники, які можуть перекриватися. Така ієрархія прямокутників є надзвичайно ефективною для індексації неточкових об'єктів (як-от полігони доріг чи будівель) і є основою багатьох сучасних просторових баз даних.

- **BSP-дерево (Binary Space Partitioning tree):** Ця структура рекурсивно ділить простір на дві частини довільною гіперплощиною. BSP-дерева є фундаментальним інструментом у 3D-графіці та відеоіграх, де вони використовуються для визначення видимості (що бачить камера), виявлення зіткнень та рендерингу складних сцен у правильному порядку.

Вивчення цих структур даних відкриє вам світ алгоритмічних технік, що лежать в основі сучасних ГІС, баз даних, комп'ютерної графіки та робототехніки.

11.4.1. Алгоритм KD-дерево

Теоретичні відомості

Основні поняття та визначення

KD-дерево (k-dimensional tree) — це бінарна структура даних для організації точок у k-вимірному просторі. Назва походить від того, що структура може працювати з даними довільної розмірності k. Кожен вузол дерева представляє k-вимірну точку і розділяє простір на дві частини гіперплощиною, перпендикулярною до однієї з осей координат.

Структура вузла KD-дерева

Кожен вузол KD-дерева містить:

- point — k-вимірна точка даних

- splitDim — вимір (вісь), за яким відбувається розділення в цьому вузлі

- left — вказівник на ліве піддерево (точки з меншим значенням у splitDim)

- right — вказівник на праве піддерево (точки з більшим або рівним значенням у splitDim)

```
STRUCTURE KDNode:
    point: Array[k]      // k-вимірна точка
    splitDim: Integer    // вимір розділення (0 до k-1)
    left: KDNode        // ліве піддерево
```

```
    right: KDNode       // праве піддерево
END STRUCTURE
```

Принцип побудови

KD-дерево будується рекурсивно, чергуючи виміри розділення на кожному рівні. На глибині d використовується вимір d mod k для розділення. Це забезпечує рівномірний розподіл розділень по всіх вимірах.

Алгоритм побудови KD-дерева такий:

```
FUNCTION BuildKDTree(points, depth):
    // Базовий випадок: немає точок
    IF points is empty:
        RETURN NULL
    END IF

    // Визначення виміру розділення
    k = dimensionality of points
    splitDim = depth MOD k

    // Сортування точок за поточним виміром
    SortPointsByDimension(points, splitDim)

    // Знаходження медіани
    medianIndex = length(points) / 2

    // Створення вузла
    node = NEW KDNode
    node.point = points[medianIndex]
    node.splitDim = splitDim

    // Рекурсивна побудова піддерев
    leftPoints = points[0 .. medianIndex-1]
    rightPoints = points[medianIndex+1 .. end]

    node.left = BuildKDTree(leftPoints, depth + 1)
    node.right = BuildKDTree(rightPoints, depth + 1)

    RETURN node
END FUNCTION
```

Розглянемо оптимізовану побудову з вибором оптимального виміру. Замість циклічного чергування вимірів, можна вибирати вимір з найбільшою дисперсією:

```
FUNCTION BuildOptimizedKDTree(points, depth):
    IF points is empty:
```

```
        RETURN NULL
    END IF

    // Знаходження виміру з максимальною дисперсією
    splitDim = FindDimensionWithMaxVariance(points)

    // Альтернатива: використання виміру з найбільшим розкидом
    // splitDim = FindDimensionWithMaxRange(points)

    // Сортування та побудова як раніше
    SortPointsByDimension(points, splitDim)
    medianIndex = length(points) / 2

    node = NEW KDNode
    node.point = points[medianIndex]
    node.splitDim = splitDim

    leftPoints = points[0 .. medianIndex-1]
    rightPoints = points[medianIndex+1 .. end]

    node.left = BuildOptimizedKDTree(leftPoints, depth + 1)
    node.right = BuildOptimizedKDTree(rightPoints, depth + 1)

    RETURN node
END FUNCTION

FUNCTION FindDimensionWithMaxVariance(points):
    k = dimensionality of points
    maxVariance = 0
    bestDim = 0

    FOR dim = 0 TO k-1:
        // Обчислення середнього значення
        sum = 0
        FOR EACH point IN points:
            sum += point[dim]
        END FOR
        mean = sum / length(points)

        // Обчислення дисперсії
        variance = 0
        FOR EACH point IN points:
            diff = point[dim] - mean
            variance += diff * diff
        END FOR
```

```
        variance /= length(points)

        IF variance > maxVariance:
            maxVariance = variance
            bestDim = dim
        END IF
    END FOR

    RETURN bestDim
END FUNCTION
```

Алгоритм пошуку найближчого сусіда

Це одна з найважливіших операцій для KD-дерева:

```
FUNCTION NearestNeighborSearch(root, target):
    // Ініціалізація
    bestNode = NULL
    bestDistance = INFINITY

    // Рекурсивний пошук
    SearchNN(root, target, 0, bestNode, bestDistance)

    RETURN bestNode
END FUNCTION

FUNCTION SearchNN(node, target, depth, bestNode, bestDistance):
    IF node is NULL:
        RETURN
    END IF

    // Обчислення відстані до поточного вузла
    distance = EuclideanDistance(node.point, target)

    // Оновлення найкращого результату
    IF distance < bestDistance:
        bestDistance = distance
        bestNode = node
    END IF

    // Визначення напрямку пошуку
    splitDim = node.splitDim

    IF target[splitDim] < node.point[splitDim]:
        // Спочатку шукаємо в лівому піддереві
        SearchNN(node.left, target, depth + 1, bestNode, bestDistance
)
```

```
        // Перевірка, чи потрібно шукати в правому піддереві
        IF abs(target[splitDim] - node.point[splitDim]) < bestDistanc
e:
            SearchNN(node.right, target, depth + 1, bestNode, bestDis
tance)
        END IF
    ELSE:
        // Спочатку шукаємо в правому піддереві
        SearchNN(node.right, target, depth + 1, bestNode, bestDistanc
e)

        // Перевірка, чи потрібно шукати в лівому піддереві
        IF abs(target[splitDim] - node.point[splitDim]) < bestDistanc
e:
            SearchNN(node.left, target, depth + 1, bestNode, bestDist
ance)
        END IF
    END IF
END FUNCTION

FUNCTION EuclideanDistance(point1, point2):
    sum = 0
    FOR i = 0 TO k-1:
        diff = point1[i] - point2[i]
        sum += diff * diff
    END FOR
    RETURN sqrt(sum)
END FUNCTION
```

Алгоритм пошуку k найближчих сусідів

```
FUNCTION KNearestNeighbors(root, target, k):
    // Використовуємо купу для зберігання k найкращих результатів
    maxHeap = NEW MaxHeap(k)  // Впорядкована за відстанню

    SearchKNN(root, target, 0, maxHeap, k)

    // Конвертуємо купу в список
    result = []
    WHILE NOT maxHeap.isEmpty():
        result.append(maxHeap.extractMax())
    END WHILE

    // Повертаємо в порядку зростання відстані
    RETURN reverse(result)
```

```
END FUNCTION

FUNCTION SearchKNN(node, target, depth, maxHeap, k):
    IF node is NULL:
        RETURN
    END IF

    // Обчислення відстані
    distance = EuclideanDistance(node.point, target)

    // Додавання до купи
    IF maxHeap.size() < k:
        maxHeap.insert(node, distance)
    ELSE IF distance < maxHeap.peekMaxDistance():
        maxHeap.extractMax()
        maxHeap.insert(node, distance)
    END IF

    // Визначення напрямку пошуку
    splitDim = node.splitDim

    IF target[splitDim] < node.point[splitDim]:
        // Спочатку ліве піддерево
        SearchKNN(node.left, target, depth + 1, maxHeap, k)

        // Перевірка правого піддерева
        IF maxHeap.size() < k OR
           abs(target[splitDim] - node.point[splitDim]) < maxHeap.pee
kMaxDistance():
            SearchKNN(node.right, target, depth + 1, maxHeap, k)
        END IF
    ELSE:
        // Спочатку праве піддерево
        SearchKNN(node.right, target, depth + 1, maxHeap, k)

        // Перевірка лівого піддерева
        IF maxHeap.size() < k OR
           abs(target[splitDim] - node.point[splitDim]) < maxHeap.pee
kMaxDistance():
            SearchKNN(node.left, target, depth + 1, maxHeap, k)
        END IF
    END IF
END FUNCTION
```

Алгоритм пошуку в діапазоні

Знаходження всіх точок у заданому прямокутному діапазоні:

```
FUNCTION RangeSearch(root, minBounds, maxBounds):
    result = []
    SearchRange(root, minBounds, maxBounds, 0, result)
    RETURN result
END FUNCTION

FUNCTION SearchRange(node, minBounds, maxBounds, depth, result):
    IF node is NULL:
        RETURN
    END IF

    // Перевірка, чи точка входить в діапазон
    inRange = TRUE
    FOR i = 0 TO k-1:
        IF node.point[i] < minBounds[i] OR node.point[i] > maxBounds[
i]:
            inRange = FALSE
            BREAK
        END IF
    END FOR

    IF inRange:
        result.append(node.point)
    END IF

    // Рекурсивний пошук в піддеревах
    splitDim = node.splitDim

    // Перевірка лівого піддерева
    IF minBounds[splitDim] <= node.point[splitDim]:
        SearchRange(node.left, minBounds, maxBounds, depth + 1, resul
t)
    END IF

    // Перевірка правого піддерева
    IF maxBounds[splitDim] >= node.point[splitDim]:
        SearchRange(node.right, minBounds, maxBounds, depth + 1, resu
lt)
    END IF
END FUNCTION
```

Алгоритм вставки нової точки

```
FUNCTION Insert(root, newPoint, depth):
    IF root is NULL:
```

```
        node = NEW KDNode
        node.point = newPoint
        node.splitDim = depth MOD k
        node.left = NULL
        node.right = NULL
        RETURN node
    END IF

    splitDim = root.splitDim

    IF newPoint[splitDim] < root.point[splitDim]:
        root.left = Insert(root.left, newPoint, depth + 1)
    ELSE:
        root.right = Insert(root.right, newPoint, depth + 1)
    END IF

    RETURN root
END FUNCTION
```

Алгоритм видалення точки

Видалення з KD-дерева є складнішою операцією:

```
FUNCTION Delete(root, targetPoint, depth):
    IF root is NULL:
        RETURN NULL
    END IF

    // Знайшли вузол для видалення
    IF PointsEqual(root.point, targetPoint):
        // Випадок 1: Лист
        IF root.left is NULL AND root.right is NULL:
            RETURN NULL
        END IF

        // Випадок 2: Є праве піддерево
        IF root.right is not NULL:
            // Знаходимо мінімум в правому піддереві за splitDim
            minNode = FindMin(root.right, root.splitDim, depth + 1)
            root.point = minNode.point
            root.right = Delete(root.right, minNode.point, depth + 1)
        ELSE:
            // Випадок 3: Тільки ліве піддерево
            // Знаходимо мінімум в лівому піддереві
            minNode = FindMin(root.left, root.splitDim, depth + 1)
            root.point = minNode.point
            // Переміщуємо ліве піддерево вправо
```

```
            root.right = root.left
            root.left = NULL
            root.right = Delete(root.right, minNode.point, depth + 1)
        END IF

        RETURN root
    END IF

    // Рекурсивний пошук вузла для видалення
    splitDim = root.splitDim

    IF targetPoint[splitDim] < root.point[splitDim]:
        root.left = Delete(root.left, targetPoint, depth + 1)
    ELSE:
        root.right = Delete(root.right, targetPoint, depth + 1)
    END IF

    RETURN root
END FUNCTION

FUNCTION FindMin(node, targetDim, depth):
    IF node is NULL:
        RETURN NULL
    END IF

    splitDim = node.splitDim

    // Якщо поточний вимір збігається з цільовим
    IF splitDim == targetDim:
        IF node.left is NULL:
            RETURN node
        ELSE:
            RETURN FindMin(node.left, targetDim, depth + 1)
        END IF
    ELSE:
        // Потрібно перевірити обидва піддерева
        leftMin = FindMin(node.left, targetDim, depth + 1)
        rightMin = FindMin(node.right, targetDim, depth + 1)

        minNode = node

        IF leftMin is not NULL AND leftMin.point[targetDim] < minNode
.point[targetDim]:
            minNode = leftMin
        END IF
```

```
        IF rightMin is not NULL AND rightMin.point[targetDim] < minNo
de.point[targetDim]:
            minNode = rightMin
        END IF

        RETURN minNode
    END IF
END FUNCTION
```

Балансування KD-дерева

Після багатьох операцій вставки/видалення дерево може стати незбалансованим:

```
FUNCTION Rebalance(root):
    // Збираємо всі точки
    points = []
    CollectPoints(root, points)

    // Перебудовуємо дерево
    RETURN BuildKDTree(points, 0)
END FUNCTION

FUNCTION CollectPoints(node, points):
    IF node is NULL:
        RETURN
    END IF

    points.append(node.point)
    CollectPoints(node.left, points)
    CollectPoints(node.right, points)
END FUNCTION
```

Область застосування

KD-дерева широко використовуються в багатьох галузях комп’ютерних наук та інженерії завдяки їх ефективності в роботі з багатовимірними даними:

- Комп’ютерний зір — пошук відповідностей між ключовими точками зображень, розпізнавання об’єктів, класифікація зображень на основі найближчих сусідів

- Машинне навчання — алгоритм k-найближчих сусідів (k-NN), кластеризація даних, пошук аномалій, зменшення розмірності

- Робототехніка — планування шляху, локалізація робота, побудова карт оточення, уникнення перешкод

- Географічні інформаційні системи — пошук найближчих об’єктів на карті (ресторани, заправки), геокодування, просторові запити

- Комп’ютерна графіка — трасування променів, виявлення зіткнень, рендеринг точкових хмар, обробка 3D-сканів

- Бази даних — індексування багатовимірних даних, прискорення просторових запитів, пошук за подібністю

- Біоінформатика — аналіз генетичних послідовностей, пошук подібних білкових структур, кластеризація генів

- Обробка сигналів — стиснення даних, квантування векторів, розпізнавання мовлення

Приклади застосування

Приклад 1: Пошук найближчої заправки

Розглянемо систему навігації, що шукає найближчу заправку до поточного положення автомобіля:

```
// Структура даних для заправки
STRUCTURE GasStation:
    location: [latitude, longitude]
    name: String
    price: Float
END STRUCTURE

// Побудова індексу заправок
gasStations = LoadGasStationsFromDatabase()
kdTreeIndex = NEW KDTree(k=2)

FOR EACH station IN gasStations:
    kdTreeIndex.insert(station.location)
END FOR

// Пошук найближчої заправки
FUNCTION FindNearestGasStation(currentLocation):
    nearestPoint = kdTreeIndex.nearestNeighbor(currentLocation)

    // Знаходимо відповідну заправку
    FOR EACH station IN gasStations:
        IF station.location == nearestPoint:
            distance = CalculateDistance(currentLocation, station.loc
```

```
ation)
            RETURN (station, distance)
        END IF
    END FOR
END FUNCTION

// Використання
currentLocation = [48.8566, 2.3522]  // Париж
(nearestStation, distance) = FindNearestGasStation(currentLocation)
PRINT "Найближча заправка: " + nearestStation.name + " на відстані "
+ distance + " км"
```

Приклад 2: Класифікація рукописних цифр методом k-NN

KD-дерево прискорює алгоритм k-найближчих сусідів для розпізнавання образів:

```
// Підготовка даних
FUNCTION PrepareDigitRecognition():
    // Завантаження навчальної вибірки (наприклад, MNIST)
    trainingData = LoadMNISTDataset()

    // Кожне зображення 28x28 перетворюємо в вектор з 784 ознак
    kdTree = NEW KDTree(k=784)

    FOR EACH sample IN trainingData:
        // Перетворення зображення в вектор
        features = FlattenImage(sample.image)
        // Зберігаємо індекс для подальшого пошуку мітки
        kdTree.insert(features, sample.index)
    END FOR

    RETURN (kdTree, trainingData)
END FUNCTION

// Класифікація нової цифри
FUNCTION ClassifyDigit(newImage, kdTree, trainingData, k):
    // Перетворення зображення в вектор ознак
    features = FlattenImage(newImage)

    // Знаходимо k найближчих сусідів
    neighbors = kdTree.kNearestNeighbors(features, k)

    // Підрахунок голосів
    votes = CreateArray(10, 0)  // Для цифр 0-9

    FOR EACH neighbor IN neighbors:
```

```
        index = neighbor.dataIndex
        label = trainingData[index].label
        votes[label] += 1
    END FOR

    // Знаходимо найпопулярнішу мітку
    predictedDigit = FindMaxIndex(votes)
    confidence = votes[predictedDigit] / k

    RETURN (predictedDigit, confidence)
END FUNCTION

// Використання
newImage = CaptureHandwrittenDigit()
(digit, confidence) = ClassifyDigit(newImage, kdTree, trainingData, k
=5)
PRINT "Розпізнана цифра: " + digit + " з впевненістю " + confidence *
100 + "%"
```

Приклад 3: Оптимізація розміщення вишок мобільного зв’язку

```
// Аналіз покриття території
FUNCTION AnalyzeCellTowerCoverage(towers, testPoints):
    // Побудова KD-дерева для швидкого пошуку
    kdTree = NEW KDTree(k=2)

    FOR EACH tower IN towers:
        kdTree.insert([tower.x, tower.y])
    END FOR

    // Аналіз покриття для кожної тестової точки
    coverageMap = []

    FOR EACH point IN testPoints:
        // Знаходимо найближчу вишку
        nearestTower = kdTree.nearestNeighbor([point.x, point.y])
        distance = CalculateDistance(point, nearestTower)

        // Оцінка якості сигналу на основі відстані
        signalStrength = CalculateSignalStrength(distance)

        coverageMap.append({
            location: point,
            tower: nearestTower,
            signalStrength: signalStrength
        })
```

```
    END FOR

    RETURN coverageMap
END FUNCTION

// Знаходження зон з поганим покриттям
FUNCTION FindDeadZones(coverageMap, minSignalThreshold):
    deadZones = []

    FOR EACH entry IN coverageMap:
        IF entry.signalStrength < minSignalThreshold:
            deadZones.append(entry.location)
        END IF
    END FOR

    // Кластеризація мертвих зон для визначення оптимальних місць нов
их вишок
    RETURN ClusterPoints(deadZones)
END FUNCTION
```

Переваги та недоліки

Переваги KD-дерева:

- Ефективність пошуку — середня складність пошуку найближчого сусіда O(log n), що значно краще за лінійний пошук O(n)

- Простота реалізації — алгоритм інтуїтивно зрозумілий та відносно простий у реалізації порівняно з іншими просторовими структурами

- Оптимальність для точкових даних — KD-дерева спеціально розроблені для роботи з точками в багатовимірному просторі

- Ефективне використання пам’яті — потребує O(n) пам’яті для зберігання n точок

- Підтримка різних типів запитів — пошук найближчих сусідів, пошук в діапазоні, k-найближчих сусідів

- Детермінованість — для одного набору даних завжди будується однакове дерево

Недоліки KD-дерева:

- Погіршення продуктивності при високій розмірності — для k > 20 ефективність різко падає через “прокляття розмірності”

- Складність динамічних операцій — вставка та видалення можуть призвести до незбалансованості дерева

- Необхідність перебудови — для підтримки збалансованості може знадобитися повна перебудова

- Неефективність для об’єктів з протяжністю — KD-дерева оптимізовані для точок, а не для прямокутників чи інших фігур

- Чутливість до розподілу даних — нерівномірний розподіл точок може призвести до незбалансованого дерева

- Складність паралелізації — ієрархічна структура ускладнює паралельну обробку

Контрольні запитання

1. Що таке KD-дерево і для чого воно використовується? Поясніть основний принцип його роботи.

2. Як відбувається вибір виміру для розділення при побудові KD-дерева? Які існують стратегії?

3. Опишіть алгоритм пошуку найближчого сусіда в KD-дереві. Чому він ефективніший за лінійний пошук?

4. Що таке “прокляття розмірності” і як воно впливає на продуктивність KD-дерева?

5. Чим відрізняється пошук k найближчих сусідів від пошуку одного найближчого сусіда?

6. Як виконується операція видалення вузла з KD-дерева? Які складнощі при цьому виникають?

7. Коли необхідна перебудова KD-дерева? Які критерії незбалансованості можна використовувати?

8. Порівняйте часову складність операцій для збалансованого та незбалансованого KD-дерева.

9. Як KD-дерево використовується для прискорення алгоритму k-найближчих сусідів у машинному навчанні?

10. Які альтернативи KD-дереву існують для роботи з високорозмірними даними?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму KD-дерева для просторового пошуку

2. Мета

Вивчити принципи роботи KD-дерева, реалізувати основні операції (побудова, пошук, вставка), дослідити ефективність структури для різних розподілів даних та розмірностей простору.

3. Завдання

- Реалізувати клас KDNode та KDTree
- Реалізувати побудову збалансованого KD-дерева
- Реалізувати пошук найближчого сусіда
- Реалізувати пошук k найближчих сусідів
- Реалізувати пошук точок у заданому діапазоні
- Реалізувати візуалізацію дерева для 2D випадку
- Провести експериментальне дослідження продуктивності

4. Опис

Лабораторна робота передбачає покрокову реалізацію KD-дерева, починаючи з базових структур даних і закінчуючи складними операціями пошуку. Студенти мають реалізувати як рекурсивні, так і ітеративні версії алгоритмів, порівняти їх ефективність на різних наборах даних.

5. Порядок виконання

- Створити структуру даних для представлення вузла KD-дерева
- Реалізувати функцію побудови дерева з масиву точок
- Реалізувати функцію виведення структури дерева
- Створити функцію пошуку найближчого сусіда
- Розширити пошук для k найближчих сусідів
- Додати функцію пошуку в прямокутному діапазоні
- Реалізувати динамічну вставку нових точок
- Створити візуалізацію для 2D випадку
- Провести тестування на різних розподілах точок
- Виміряти час виконання операцій для різних розмірів даних

6. Результат

Програма з повною реалізацією KD-дерева, візуалізацією для двовимірного випадку, набором тестів та аналізом продуктивності. Графіки залежності часу виконання операцій від кількості точок та розмірності простору.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас KDNode з полями для точки, виміру розділення та вказівників на піддерева

- Клас KDTree з методами build(), insert(), nearestNeighbor(), kNearestNeighbors(), rangeSearch()

- Функції для обчислення відстані та порівняння точок

- Модуль візуалізації для відображення дерева та процесу пошуку

2. Продемонструвати роботу додатку:

- Побудова дерева з випадкових точок

- Візуалізація структури дерева з розділюючими лініями

- Демонстрація пошуку найближчого сусіда з підсвічуванням шляху пошуку

- Показ роботи пошуку k найближчих сусідів

- Демонстрація пошуку в діапазоні з візуалізацією області

- Графіки продуктивності для різних операцій

3. Бути готовим відповісти на питання:

- Як вибирається медіана при побудові дерева?

- Чому важливо перевіряти обидва піддерева при пошуку найближчого сусіда?

- Як змінюється продуктивність при збільшенні розмірності?

- Які оптимізації можна застосувати для покращення швидкодії?

- Як реалізувати видалення точки зі збереженням збалансованості?

Тести

1. KD-дерево розділяє простір:

1) Випадковими площинами

2) Гіперплощинами, перпендикулярними до осей координат

3) Сферичними поверхнями

4) Діагональними площинами

2. При побудові KD-дерева на глибині d використовується вимір:

1) d

2) d - 1

3) d mod k

4) k - d

3. Середня складність пошуку найближчого сусіда в збалансованому KD-дереві:

1) O(n)

2) O(log n)

3) O(n log n)

4) O(1)

4. Для видалення вузла з KD-дерева потрібно знайти:

1) Максимум у лівому піддереві

2) Мінімум у правому піддереві за виміром розділення

3) Будь-який лист дерева

4) Корінь дерева

5. “Прокляття розмірності” для KD-дерева проявляється при:

1) k > 2

2) k > 10

3) k > 20

4) k > 100

Відповіді:

1-2,

2-3,

3-2,

4-2,

5-3

Тести підвищеної складності

1. При якому розподілі точок KD-дерево буде максимально незбалансованим?

1) Рівномірний розподіл у квадраті

2) Нормальний розподіл

3) Точки на одній прямій

4) Точки у вершинах правильного многокутника

2. Яка структура даних найкраще підходить для реалізації k найближчих сусідів у KD-дереві?

1) Стек

2) Черга

3) Купа максимумів

4) Хеш-таблиця

3. Складність побудови KD-дерева з n точок у k-вимірному просторі:

1) $O(n\ log\ n)$

2) $O(kn\ log\ n)$

3) $O(n^2)$

4) $O(kn^2)$

4. Коли при пошуку найближчого сусіда потрібно перевіряти обидва піддерева вузла?

1) Завжди

2) Ніколи

3) Коли відстань до розділюючої гіперплощини менша за поточну найкращу відстань

4) Коли цільова точка лежить на розділюючій гіперплощині

5. Який метод вибору виміру розділення дає найкраще збалансоване дерево?

1) Циклічний вибір

2) Випадковий вибір

3) Вибір виміру з максимальною дисперсією

4) Вибір виміру з мінімальною дисперсією

6. Для прискорення пошуку в KD-дереві можна використовувати:

1) Кешування останніх запитів

2) Апроксимацію відстані

3) Паралельний пошук у піддеревах

4) Всі перераховані методи

7. При якій кількості точок варто перейти від KD-дерева до лінійного пошуку?

1) Ніколи

2) При $n < 10$

3) При $n < k^2$

4) При $n < 2^k$

8. Яка модифікація KD-дерева краще працює з динамічними даними?

1) KD-дерево з відкладеним видаленням

2) Збалансоване KD-дерево з ротаціями

3) KD-дерево з буферизацією вставок

4) Адаптивне KD-дерево

9. Чому KD-дерево погано працює для пошуку об’єктів з протяжністю?

1) Занадто велика глибина дерева

2) Об’єкт може перетинати багато регіонів розділення

3) Неможливо обчислити відстань до об’єкта

4) Потрібно більше пам’яті

10. Який підхід дозволяє ефективно використовувати KD-дерево для даних, що часто змінюються?

1) Повна перебудова при кожній зміні

2) Локальна перебудова піддерев

3) Використання декількох KD-дерев з ротацією

4) Лінива перебудова з порогом незбалансованості

Відповіді:

1-3. Коли всі точки лежать на одній прямій, дерево вироджується в лінійну структуру.

2-3. Купа максимумів дозволяє ефективно зберігати k найменших відстаней.

3-2. Потрібно O(n log n) для сортування за кожним виміром, помножене на k вимірів.

4-3. Це умова гарантує, що в іншому піддереві може бути ближча точка.

5-3. Максимальна дисперсія забезпечує найкраще розділення точок.

6-4. Всі методи можуть покращити продуктивність залежно від сценарію.

7-5. При експоненційному зростанні простору пошуку лінійний метод стає ефективнішим.

8-3. Буферизація дозволяє відкладати перебудову до накопичення критичної маси змін.

9-2. Прямокутник може перетинати багато регіонів, що вимагає перевірки багатьох гілок.

10-4. Перебудова тільки при досягненні порогу незбалансованості оптимізує продуктивність.

Завдання для самостійної роботи

1. Реалізуйте алгоритм побудови KD-дерева без рекурсії, використовуючи стек. Порівняйте продуктивність з рекурсивною версією.

2. Розробіть алгоритм approximate nearest neighbor search, що знаходить сусіда з точністю $(1 + \varepsilon)$ за час $O(log\ n\ /\ \varepsilon^d)$.

3. Реалізуйте KD-дерево, що зберігає історію всіх змін і дозволяє виконувати запити для попередніх версій.

4. Створіть паралельну версію побудови KD-дерева з використанням OpenMP або потоків. Дослідить прискорення для різної кількості ядер.

5. Реалізуйте алгоритм пакетного завантаження для KD-дерева, оптимізований для побудови з великого масиву точок.

6. Розробіть адаптивне KD-дерево, що автоматично вибирає оптимальну стратегію розділення на основі локального розподілу точок.

7. Створіть гібридну структуру, що поєднує KD-дерево для глобального розділення та інші структури (наприклад, R-дерево) для локальних областей.

8. Реалізуйте KD-дерево для роботи з точками на сфері, враховуючи особливості сферичної геометрії.

9. Розробіть систему автоматичного вибору між KD-деревом та іншими структурами даних на основі характеристик набору даних.

10. Створіть розподілене KD-дерево для роботи з даними, що не поміщаються в пам’ять одного комп’ютера.

Теми лабораторних робіт

1. Базова реалізація KD-дерева для двовимірних точок з візуалізацією процесу побудови та пошуку.

2. Порівняльний аналіз різних стратегій вибору виміру розділення (циклічна, за дисперсією, за діапазоном).

3. Реалізація та оптимізація алгоритму k-найближчих сусідів з використанням KD-дерева для класифікації зображень.

4. Розробка системи геопросторового пошуку з використанням KD-дерева для індексації об’єктів на карті.

5. Дослідження впливу розмірності на продуктивність KD-дерева, пошук оптимальних параметрів для різних k.

6. Створення динамічного KD-дерева з підтримкою ефективної вставки та видалення точок.

7. Реалізація променевого перетину з використанням KD-дерева для прискорення пошуку перетинів.

8. Розробка системи пошуку схожих документів з використанням KD-дерева в просторі TF-IDF векторів.

9. Паралельна реалізація операцій KD-дерева з використанням GPU (CUDA/OpenCL).

10. Створення адаптивної системи вибору просторової структури даних (KD-дерево проти R-дерева та Ball-дерева) на основі аналізу даних.

11.4.1. Алгоритм R-дерево

Теоретичні відомості

Основні поняття та визначення

R-дерево (R-tree) — це збалансована ієрархічна структура даних для індексування просторових об’єктів, розроблена Антоніном Гуттманом у 1984 році. На відміну від KD-дерева, яке працює з точками, R-дерево спеціально призначене для роботи з об’єктами, що мають просторову протяжність — прямокутниками, полігонами, колами та іншими геометричними фігурами. Назва походить від слова “Rectangle”, оскільки основним елементом структури є мінімальний обмежуючий прямокутник (MBR - Minimum Bounding Rectangle).

Структура R-дерева

1. **Ієрархія прямокутників:** R-дерево організовує дані в ієрархічну структуру.

- Листкові вузли: Містять записи форми `(MBR, obj_id)`, де `obj_id` — це ідентифікатор реального геометричного об'єкта, а `MBR` — це найменший прямокутник, що повністю його охоплює.

- Внутрішні вузли: Містять записи форми `(MBR, child_ptr)`, де `child_ptr` — це вказівник на дочірній вузол, а `MBR` — це прямокутник, що охоплює всі прямокутники у цьому дочірньому вузлі.

2. **Параметри `m` та `M`:** Кожен вузол (крім, можливо, кореня) повинен містити від `m` до `M` записів, де $m \leq M/2$. Це гарантує збалансованість дерева та ефективне використання простору.

3. **Збалансованість:** Усі листкові вузли знаходяться на однаковій глибині.

4. **Перекриття:** MBR дочірніх вузлів одного батька можуть перекриватися. Мінімізація цього перекриття є головною метою при побудові та модифікації дерева.

Уявіть карту міста.

- Кожен будинок (об'єкт) знаходиться у листковому вузлі.

- Групи сусідніх будинків (наприклад, квартал) об'єднуються в MBR у батьківському вузлі.

- Ці квартали, у свою чергу, групуються в MBR районів на ще вищому рівні, і так до MBR, що охоплює все місто (корінь дерева).

Перед тим, як почнемо вивчати R-дерева, все ж таки розглянемо їхню сутність та необхідність.

R-дерево потрібне для ефективного вирішення специфічної проблеми: **швидкого пошуку просторових об'єктів, які мають геометричну протяжність**.

Уявіть, що у вас є база даних з мільйоном будівель на карті. Кожна будівля - це не точка, а полігон з площею. Вам потрібно знайти всі будівлі в радіусі 1 км від певної локації.

Без R-дерева: потрібно перевірити кожну з мільйона будівель - O(n) операцій.

З R-деревом: потрібно перевірити лише кілька гілок дерева - O(log n) операцій.

Давайте розглянемо також інші структури данних.

KD-дерево

- Працює тільки з **точками**, а не з об’єктами, що мають площу
- Для прямокутника довелося б зберігати всі його кути як окремі точки
- Неефективно для полігонів та інших складних фігур

Звичайні бази даних (B-дерева індекси)

- Можуть індексувати тільки одновимірні дані
- Для координат (x, y) довелося б створювати два окремі індекси
- Пошук “всіх об’єктів в області” вимагав би складних об’єднань результатів

Сітка

- Потребує багато пам’яті для розріджених даних
- Погано працює з об’єктами різних розмірів
- Неефективна при нерівномірному розподілі об’єктів

Наведемо реальні приклади, де R-дерево незамінне:

1. Карти в смартфоні

Коли ви відкриваєте Google Maps і наближаєте певну область, додаток повинен швидко знайти всі об’єкти (будівлі, дороги, POI), які потрапляють у видиму область екрану. R-дерево дозволяє це зробити за долі секунди.

2. Доставка їжі

Служби доставки використовують R-дерева для пошуку ресторанів у зоні доставки. Зона доставки - це складний полігон, і потрібно знайти всі ресторани, що в нього потрапляють.

3. Комп’ютерні ігри

У грі з тисячами об’єктів потрібно перевіряти зіткнення тільки між тими, що знаходяться поруч. R-дерево групує близькі об’єкти, дозволяючи перевіряти лише потенційні зіткнення.

4. CAD системи

При проектуванні складних механізмів потрібно швидко знаходити деталі, що перетинаються або знаходяться в певній області креслення.

Ключові переваги R-дерева:

1. Працює з об’єктами будь-якої форми - зберігає мінімальний обмежуючий прямокутник

2. Динамічне - можна додавати/видаляти об’єкти без перебудови всієї структури

3. Збалансоване - гарантує логарифмічну складність операцій

4. Ефективне для дискової пам’яті - мінімізує кількість звернень до диска

Тому R-дерево - це не просто “ще одна структура даних”, а критично важливий інструмент для всіх систем, що працюють з геопросторовими даними або геометричними об’єктами. Без нього сучасні карти, ГІС-системи та багато ігор працювали б неприйнятно повільно.

Тепер перейдемо до вивчення структури вузла R-дерева:

```
STRUCTURE RTreeNode:
    level: Integer              // Рівень вузла (0 для листів)
    count: Integer              // Кількість записів у вузлі
    entries: Array[M] of Entry  // Масив записів
    parent: RTreeNode           // Вказівник на батьківський вузол
END STRUCTURE

STRUCTURE Entry:
    mbr: Rectangle              // Мінімальний обмежуючий прямокутник
    child: UNION                // Залежить від типу вузла:
        object: SpatialObject   // Для листових вузлів - об'єкт даних
        node: RTreeNode         // Для внутрішніх вузлів - дочірній в
узол
    END UNION
END STRUCTURE

STRUCTURE Rectangle:
    min: Array[d] of Float      // Мінімальні координати по кожному в
иміру
    max: Array[d] of Float      // Максимальні координати по кожному
виміру
END STRUCTURE
```

Властивості R-дерева

R-дерево має наступні важливі властивості:

- Кожен листовий вузол містить від m до M записів про просторові об’єкти

- Кожен внутрішній вузол містить від m до M записів про дочірні вузли

- Корінь містить принаймні 2 дочірні вузли (якщо він не лист)

- Всі листи знаходяться на одному рівні (дерево збалансоване)

- MBR кожного внутрішнього вузла є найменшим прямокутником, що містить всі MBR його дочірніх вузлів

- MBR можуть перекриватися, що відрізняє R-дерево від багатьох інших просторових структур

Обчислення MBR

Розглянемо алгоритм обчислення мінімального обмежуючого прямокутника для набору об’єктів:

```
FUNCTION CalculateMBR(objects):
    IF objects is empty:
        RETURN NULL
    END IF

    // Ініціалізація першим об'єктом
    mbr = NEW Rectangle
    firstObject = objects[0]

    FOR dim = 0 TO d-1:
        mbr.min[dim] = GetMinCoordinate(firstObject, dim)
        mbr.max[dim] = GetMaxCoordinate(firstObject, dim)
    END FOR

    // Розширення для включення всіх об'єктів
    FOR i = 1 TO length(objects)-1:
        object = objects[i]
        FOR dim = 0 TO d-1:
            mbr.min[dim] = MIN(mbr.min[dim], GetMinCoordinate(object,
dim))
            mbr.max[dim] = MAX(mbr.max[dim], GetMaxCoordinate(object,
dim))
        END FOR
    END FOR

    RETURN mbr
END FUNCTION
```

Пошук в R-дереві

Розглянемо алгоритм пошуку всіх об’єктів, що перетинаються з заданим прямокутником. Пошук є найпростішою операцією. Мета — знайти всі об’єкти, що перетинаються із заданим пошуковим прямокутником S.

- **Крок 1:** Почати з кореневого вузла.

- **Крок 2:** Для кожного запису (`MBR, ptr`) у поточному вузлі перевірити, чи перетинається `MBR` з `S`.

- **Крок 3:** Для всіх записів, де `MBR` перетинається з `S`:

- Якщо поточний вузол — внутрішній, рекурсивно повторити пошук у дочірньому вузлі, на який вказує `ptr`.

- Якщо поточний вузол — листковий, перевірити, чи перетинається сам об’єкт (на який вказує `ptr`) з `S`. Якщо так, додати об’єкт до результатів.

```
FUNCTION Search(root, searchRectangle):
    result = []
    SearchRecursive(root, searchRectangle, result)
    RETURN result
END FUNCTION

FUNCTION SearchRecursive(node, searchRectangle, result):
    IF node is NULL:
        RETURN
    END IF

    FOR i = 0 TO node.count-1:
        entry = node.entries[i]

        // Перевірка перетину MBR
        IF Intersects(entry.mbr, searchRectangle):
            IF node.level == 0:  // Листовий вузол
                // Додаємо об'єкт до результату
                result.append(entry.child.object)
            ELSE:  // Внутрішній вузол
                // Рекурсивний пошук у дочірньому вузлі
                SearchRecursive(entry.child.node, searchRectangle, re
sult)
            END IF
        END IF
    END FOR
END FUNCTION

FUNCTION Intersects(rect1, rect2):
    FOR dim = 0 TO d-1:
        IF rect1.max[dim] < rect2.min[dim] OR rect1.min[dim] > rect2.
max[dim]:
            RETURN FALSE
        END IF
```

```
    END FOR
    RETURN TRUE
END FUNCTION
```

Вставка в R-дерево

Алгоритм вставки нового об’єкта в R-дерево складається з декількох етапів. Вставка є значно складнішою операцією, оскільки вона може вимагати розширення MBR та розділення вузлів.

- **Фаза 1: Вибір піддерева.**

- Починаючи з кореня, на кожному рівні потрібно обрати, в який дочірній вузол вставити новий об’єкт. Для цього використовується евристика: обирається той вузол, чий MBR потребує найменшого розширення площі, щоб включити MBR нового об’єкта. При рівних значеннях обирається вузол з MBR найменшої площі.

- Цей процес повторюється, доки не буде досягнуто листкового вузла.

- **Фаза 2: Вставка у листковий вузол.**

- Якщо у знайденому листковому вузлі є вільне місце (кількість записів `< M`), новий об’єкт просто додається до нього.

- Якщо вузол заповнений (кількість записів `== M`), відбувається переповнення, і вузол необхідно розділити.

- **Фаза 3: Розділення вузла.**

- Це найскладніша частина. Мета — розділити `M+1` записів на два нові вузли так, щоб мінімізувати площу та перекриття їхніх нових MBR.

- **Квадратичний алгоритм розділення:**

1. Знайти два записи, які є “найгіршою парою” для розміщення в одному вузлі (тобто їхній спільний MBR має найбільшу “марну” площу). Зробити їх “насінням” для двох нових вузлів.

2. Послідовно розподілити решту записів. Для кожного запису обрати той вузол, додавання до якого призведе до мінімального розширення MBR.

- Існують також лінійні та експоненційні алгоритми розділення.

- **Фаза 4: Коригування дерева.**

- Після вставки у листковий вузол (або його розділення) необхідно оновити MBR усіх батьківських вузлів на шляху до кореня.

- Якщо розділення вузла призвело до появи нового вузла, запис про нього потрібно додати до батьківського вузла. Це може, у свою чергу, спричинити

його переповнення та розділення, яке може поширюватися (propagate) вгору аж до кореня. Якщо розділяється корінь, створюється новий корінь, і висота дерева збільшується на 1.

Приклад роботи алгоритму вставки

Уявімо R-дерево з `M=3`. Ми вставляємо прямокутники R1, R2, R3, R4.

1. **Вставка R1, R2, R3:** Створюється кореневий вузол (який є і листком), і в нього додаються R1, R2, R3. Його MBR охоплює всі три прямокутники.

2. **Вставка R4:** Листковий вузол (корінь) тепер має 4 записи, що є переповненням (`> M`). Потрібно його розділити.

- **Розділення:** Алгоритм розділення (напр., квадратичний) групує записи. Припустимо, він створює два нові вузли: L1 (з R1, R2) та L2 (з R3, R4).

- **Новий корінь:** Оскільки старий корінь розділився, створюється новий кореневий вузол. Він буде внутрішнім вузлом і міститиме два записи: `(MBR(L1), ptr_L1)` та `(MBR(L2), ptr_L2)`.

Робота алгоритму така: до та після розділення:

- **До:** `[R1] [R2] [R3] [R4]` (Один вузол)

- **Після:** `Корінь -> [L1(R1,R2)], [L2(R3,R4)]` (Три вузли: один корінь, два листки)

```
FUNCTION Insert(root, newObject):
    // Створення нового запису
    newEntry = NEW Entry
    newEntry.mbr = CalculateMBR([newObject])
    newEntry.child.object = newObject

    // Вибір листа для вставки
    leaf = ChooseLeaf(root, newEntry)

    // Вставка в лист
    IF leaf.count < M:
        // Є місце - просто додаємо
        AddEntry(leaf, newEntry)
    ELSE:
        // Потрібне розщеплення
        newLeaf = SplitNode(leaf, newEntry)
        AdjustTree(leaf, newLeaf)
    END IF
END FUNCTION
```

```
FUNCTION ChooseLeaf(node, newEntry):
    current = node

    WHILE current.level > 0:  // Поки не дійшли до листа
        // Вибираємо дочірній вузол з мінімальним розширенням
        bestChild = NULL
        minEnlargement = INFINITY

        FOR i = 0 TO current.count-1:
            child = current.entries[i]
            enlargement = CalculateEnlargement(child.mbr, newEntry.mb
r)

            IF enlargement < minEnlargement OR
               (enlargement == minEnlargement AND Area(child.mbr) < A
rea(bestChild.mbr)):
                minEnlargement = enlargement
                bestChild = child
            END IF
        END FOR

        current = bestChild.child.node
    END WHILE

    RETURN current
END FUNCTION

FUNCTION CalculateEnlargement(existingMBR, newMBR):
    enlargedMBR = NEW Rectangle

    FOR dim = 0 TO d-1:
        enlargedMBR.min[dim] = MIN(existingMBR.min[dim], newMBR.min[d
im])
        enlargedMBR.max[dim] = MAX(existingMBR.max[dim], newMBR.max[d
im])
    END FOR

    RETURN Area(enlargedMBR) - Area(existingMBR)
END FUNCTION

FUNCTION Area(rect):
    area = 1.0
    FOR dim = 0 TO d-1:
        area *= (rect.max[dim] - rect.min[dim])
    END FOR
```

```
    RETURN area
END FUNCTION
```

Розщеплення вузла

Коли вузол переповнюється, його потрібно розщепити на два. Існує кілька алгоритмів розщеплення. Розглянемо квадратичний алгоритм:

```
FUNCTION SplitNode(node, newEntry):
    // Всі записи включаючи новий
    allEntries = []
    FOR i = 0 TO node.count-1:
        allEntries.append(node.entries[i])
    END FOR
    allEntries.append(newEntry)

    // Вибір початкових записів для двох груп
    (seed1, seed2) = PickSeeds(allEntries)

    group1 = [seed1]
    group2 = [seed2]

    // Видалення початкових записів зі списку
    allEntries.remove(seed1)
    allEntries.remove(seed2)

    // Розподіл решти записів
    WHILE allEntries is not empty:
        IF group1.length + allEntries.length == m:
            // Потрібно додати всі що залишились до group1
            group1.extend(allEntries)
            BREAK
        END IF

        IF group2.length + allEntries.length == m:
            // Потрібно додати всі що залишились до group2
            group2.extend(allEntries)
            BREAK
        END IF

        // Вибір наступного запису
        next = PickNext(allEntries, group1, group2)

        // Визначення до якої групи додати
        mbr1 = CalculateMBR(group1)
        mbr2 = CalculateMBR(group2)
```

```
        enlargement1 = CalculateEnlargement(mbr1, next.mbr)
        enlargement2 = CalculateEnlargement(mbr2, next.mbr)

        IF enlargement1 < enlargement2:
            group1.append(next)
        ELSE IF enlargement2 < enlargement1:
            group2.append(next)
        ELSE:
            // Однакове розширення - вибираємо меншу групу
            IF group1.length <= group2.length:
                group1.append(next)
            ELSE:
                group2.append(next)
            END IF
        END IF

        allEntries.remove(next)
    END WHILE

    // Створення нового вузла
    newNode = NEW RTreeNode
    newNode.level = node.level

    // Заповнення старого та нового вузлів
    node.count = 0
    FOR entry IN group1:
        node.entries[node.count] = entry
        node.count += 1
    END FOR

    newNode.count = 0
    FOR entry IN group2:
        newNode.entries[newNode.count] = entry
        newNode.count += 1
    END FOR

    RETURN newNode
END FUNCTION

FUNCTION PickSeeds(entries):
    maxWaste = -INFINITY
    seed1 = NULL
    seed2 = NULL

    FOR i = 0 TO length(entries)-2:
```

```
        FOR j = i+1 TO length(entries)-1:
            // Обчислення "марнування" простору
            combinedMBR = CombineMBR(entries[i].mbr, entries[j].mbr)
            waste = Area(combinedMBR) - Area(entries[i].mbr) - Area(e
ntries[j].mbr)

            IF waste > maxWaste:
                maxWaste = waste
                seed1 = entries[i]
                seed2 = entries[j]
            END IF
        END FOR
    END FOR

    RETURN (seed1, seed2)
END FUNCTION
```

Коригування дерева після розщеплення

```
FUNCTION AdjustTree(node, newNode):
    n = node
    nn = newNode

    WHILE n != root:
        parent = n.parent

        // Оновлення MBR батьківського запису
        FOR i = 0 TO parent.count-1:
            IF parent.entries[i].child.node == n:
                parent.entries[i].mbr = CalculateMBR(GetAllMBRs(n))
                BREAK
            END IF
        END FOR

        // Якщо було розщеплення, потрібно додати новий вузол
        IF nn != NULL:
            newEntry = NEW Entry
            newEntry.mbr = CalculateMBR(GetAllMBRs(nn))
            newEntry.child.node = nn
            nn.parent = parent

            IF parent.count < M:
                AddEntry(parent, newEntry)
                nn = NULL  // Розщеплення не поширюється далі
            ELSE:
                // Батьківський вузол теж потрібно розщепити
```

```
                nn = SplitNode(parent, newEntry)
            END IF
        END IF

        n = parent
    END WHILE

    // Якщо дійшли до кореня і він розщепився
    IF nn != NULL:
        // Створюємо новий корінь
        newRoot = NEW RTreeNode
        newRoot.level = root.level + 1

        // Додаємо старий та новий корені як дочірні
        entry1 = NEW Entry
        entry1.mbr = CalculateMBR(GetAllMBRs(root))
        entry1.child.node = root

        entry2 = NEW Entry
        entry2.mbr = CalculateMBR(GetAllMBRs(nn))
        entry2.child.node = nn

        newRoot.entries[0] = entry1
        newRoot.entries[1] = entry2
        newRoot.count = 2

        root.parent = newRoot
        nn.parent = newRoot

        root = newRoot
    END IF
END FUNCTION
```

Видалення з R-дерева

Алгоритм видалення об’єкта з R-дерева:

```
FUNCTION Delete(root, targetObject):
    // Знаходимо лист, що містить об'єкт
    leaf = FindLeaf(root, targetObject)

    IF leaf == NULL:
        RETURN FALSE  // Об'єкт не знайдено
    END IF

    // Видаляємо запис з листа
    FOR i = 0 TO leaf.count-1:
```

```
        IF leaf.entries[i].child.object == targetObject:
            // Зсуваємо записи
            FOR j = i TO leaf.count-2:
                leaf.entries[j] = leaf.entries[j+1]
            END FOR
            leaf.count -= 1
            BREAK
        END IF
    END FOR

    // Перевіряємо недозаповнення
    CondenseTree(leaf)

    // Якщо корінь має лише один дочірній вузол, робимо його новим ко
ренем
    IF root.count == 1 AND root.level > 0:
        root = root.entries[0].child.node
        root.parent = NULL
    END IF

    RETURN TRUE
END FUNCTION

FUNCTION FindLeaf(node, targetObject):
    IF node.level == 0:  // Це лист
        FOR i = 0 TO node.count-1:
            IF node.entries[i].child.object == targetObject:
                RETURN node
            END IF
        END FOR
        RETURN NULL
    ELSE:  // Внутрішній вузол
        FOR i = 0 TO node.count-1:
            IF Contains(node.entries[i].mbr, targetObject):
                result = FindLeaf(node.entries[i].child.node, targetO
bject)
                IF result != NULL:
                    RETURN result
                END IF
            END IF
        END FOR
        RETURN NULL
    END IF
END FUNCTION
```

```
FUNCTION CondenseTree(node):
    n = node
    eliminatedNodes = []

    WHILE n != root:
        parent = n.parent

        IF n.count < m:
            // Вузол недозаповнений - видаляємо його
            // Видаляємо запис про n з батьківського вузла
            FOR i = 0 TO parent.count-1:
                IF parent.entries[i].child.node == n:
                    FOR j = i TO parent.count-2:
                        parent.entries[j] = parent.entries[j+1]
                    END FOR
                    parent.count -= 1
                    BREAK
                END IF
            END FOR

            eliminatedNodes.append(n)
        ELSE:
            // Просто оновлюємо MBR
            FOR i = 0 TO parent.count-1:
                IF parent.entries[i].child.node == n:
                    parent.entries[i].mbr = CalculateMBR(GetAllMBRs(n
))
                    BREAK
                END IF
            END FOR
        END IF

        n = parent
    END WHILE

    // Перевставляємо записи з видалених вузлів
    FOR node IN eliminatedNodes:
        FOR i = 0 TO node.count-1:
            IF node.level == 0:
                Insert(root, node.entries[i].child.object)
            ELSE:
                // Для внутрішніх вузлів потрібна спеціальна процедур
a
                ReinsertSubtree(root, node.entries[i].child.node)
            END IF
```

```
        END FOR
    END FOR
END FUNCTION
```

Варіанти R-дерева

Існує кілька модифікацій базового алгоритму R-дерева, спрямованих на покращення продуктивності:

1. R*-дерево використовує більш складні критерії для вибору вузла при вставці та розщепленні:

```
FUNCTION ChooseSubtreeRStar(node, newEntry):
    current = node

    WHILE current.level > 1:  // До передостаннього рівня
        IF current.level == targetLevel + 1:
            // Для листового рівня - мінімізуємо перекриття
            bestChild = ChooseByMinOverlap(current, newEntry)
        ELSE:
            // Для вищих рівнів - мінімізуємо розширення площі
            bestChild = ChooseByMinEnlargement(current, newEntry)
        END IF

        current = bestChild
    END WHILE

    RETURN current
END FUNCTION

FUNCTION ChooseByMinOverlap(node, newEntry):
    minOverlap = INFINITY
    bestEntry = NULL

    FOR i = 0 TO node.count-1:
        entry = node.entries[i]

        // Обчислюємо перекриття з іншими записами
        overlap = 0
        enlargedMBR = CombineMBR(entry.mbr, newEntry.mbr)

        FOR j = 0 TO node.count-1:
            IF i != j:
                overlap += OverlapArea(enlargedMBR, node.entries[j].m
br)
                overlap -= OverlapArea(entry.mbr, node.entries[j].mbr
)
```

```
            END IF
        END FOR

        IF overlap < minOverlap:
            minOverlap = overlap
            bestEntry = entry
        END IF
    END FOR

    RETURN bestEntry.child.node
END FUNCTION
```

2. R+-дерево забороняє перекриття MBR на одному рівні, що може призвести до дублювання об’єктів у різних гілках:

```
FUNCTION InsertRPlus(root, newObject):
    // Знаходимо всі листи, чиї MBR перетинаються з новим об'єктом
    overlappingLeaves = FindOverlappingLeaves(root, newObject.mbr)

    IF overlappingLeaves is empty:
        // Звичайна вставка
        Insert(root, newObject)
    ELSE:
        // Розбиваємо об'єкт на частини
        parts = SplitObject(newObject, overlappingLeaves)

        FOR part IN parts:
            InsertPart(root, part)
        END FOR
    END IF
END FUNCTION
```

Область застосування

R-дерево знайшло широке застосування в системах, що працюють з просторовими даними завдяки своїй ефективності при роботі з об’єктами, які мають геометричну протяжність:

- Географічні інформаційні системи (GIS) — індексування географічних об’єктів таких як будівлі, дороги, річки, адміністративні кордони для швидкого пошуку об’єктів у заданій області карти

- Системи управління базами даних — просторові розширення СУБД (PostgreSQL з PostGIS, Oracle Spatial, MySQL Spatial) використовують R-дерева для індексування геометричних даних

- Комп’ютерне проектування (CAD) — пошук елементів креслення в заданій області, виявлення перетинів між деталями, оптимізація розміщення компонентів

- Комп’ютерна графіка та ігрові движки — виявлення зіткнень між об’єктами, відсікання невидимих об’єктів (frustum culling), просторове розділення сцени для оптимізації рендерингу

- Мультимедійні бази даних — індексування зображень та відео за просторовими ознаками, пошук схожих медіа-об’єктів

- Телекомунікації — оптимізація розміщення базових станцій, аналіз покриття мережі, маршрутизація в мобільних мережах

- Транспортні системи — моніторинг руху транспорту, пошук найближчих транспортних засобів, оптимізація маршрутів доставки

- Астрономія та космічні дослідження — каталогізація небесних об’єктів, пошук об’єктів у заданій області неба

Приклади застосування.

Розглянемо не тільки приклади використання, але й ще реалізацію деяких методів.

Приклад 1: Система моніторингу транспорту в місті

Розглянемо систему відстеження громадського транспорту, яка повинна ефективно обробляти запити про транспортні засоби в певній області міста:

```
// Структура для представлення транспортного засобу
STRUCTURE Vehicle:
    id: String
    type: String  // bus, tram, trolleybus
    position: Point
    boundingBox: Rectangle  // Область навколо транспорту
    route: String
    speed: Float
    lastUpdate: Timestamp
END STRUCTURE

// Система моніторингу транспорту
CLASS TransportMonitoringSystem:
    rtree: RTree
    vehicles: HashMap<String, Vehicle>

    FUNCTION UpdateVehiclePosition(vehicleId, newPosition):
        vehicle = vehicles[vehicleId]
```

```
        IF vehicle != NULL:
            // Видаляємо старе положення з R-дерева
            rtree.delete(vehicle.boundingBox, vehicleId)

            // Оновлюємо положення
            vehicle.position = newPosition
            vehicle.boundingBox = CalculateBoundingBox(newPosition, V
EHICLE_SIZE)
            vehicle.lastUpdate = CurrentTime()

            // Вставляємо нове положення
            rtree.insert(vehicle.boundingBox, vehicleId)
        END IF
    END FUNCTION

    FUNCTION FindVehiclesInArea(searchArea):
        // Пошук всіх транспортних засобів у заданій області
        vehicleIds = rtree.search(searchArea)
        result = []

        FOR EACH id IN vehicleIds:
            vehicle = vehicles[id]
            IF vehicle != NULL AND IsActive(vehicle):
                result.append(vehicle)
            END IF
        END FOR

        RETURN result
    END FUNCTION

    FUNCTION FindNearestVehicle(position, vehicleType):
        // Розширюємо область пошуку поступово
        searchRadius = 100  // метрів

        WHILE searchRadius < MAX_SEARCH_RADIUS:
            searchArea = CreateRectangle(
                position.x - searchRadius,
                position.y - searchRadius,
                position.x + searchRadius,
                position.y + searchRadius
            )

            candidates = FindVehiclesInArea(searchArea)
```

```
            // Фільтруємо за типом та знаходимо найближчий
            nearestVehicle = NULL
            minDistance = INFINITY

            FOR EACH vehicle IN candidates:
                IF vehicle.type == vehicleType:
                    distance = CalculateDistance(position, vehicle.po
sition)
                    IF distance < minDistance:
                        minDistance = distance
                        nearestVehicle = vehicle
                    END IF
                END IF
            END FOR

            IF nearestVehicle != NULL:
                RETURN nearestVehicle
            END IF

            searchRadius *= 2  // Подвоюємо радіус пошуку
        END WHILE

        RETURN NULL
    END FUNCTION

    FUNCTION AnalyzeTrafficDensity(area, timeWindow):
        // Аналіз щільності транспорту в області за період часу
        vehicles = FindVehiclesInArea(area)

        densityMap = CreateGrid(area, CELL_SIZE)

        FOR EACH vehicle IN vehicles:
            // Отримуємо історію руху за timeWindow
            trajectory = GetVehicleTrajectory(vehicle.id, timeWindow)

            FOR EACH position IN trajectory:
                cell = GetGridCell(densityMap, position)
                cell.count += 1
            END FOR
        END FOR

        RETURN densityMap
    END FUNCTION
END CLASS
```

```
// Використання системи
transportSystem = NEW TransportMonitoringSystem()

// Оновлення положення автобуса
transportSystem.UpdateVehiclePosition("bus_123", Point(48.464, 35.046
))

// Пошук автобусів поблизу зупинки
busStop = Point(48.465, 35.048)
searchArea = Rectangle(busStop.x - 0.005, busStop.y - 0.005,
                       busStop.x + 0.005, busStop.y + 0.005)
nearbyBuses = transportSystem.FindVehiclesInArea(searchArea)

PRINT "Знайдено " + nearbyBuses.length + " автобусів поблизу зупинки"
```

Приклад 2: Система виявлення зіткнень у грі

R-дерево ефективно використовується для оптимізації виявлення зіткнень між ігровими об’єктами:

```
// Ігровий об'єкт з фізикою
STRUCTURE GameObject:
    id: Integer
    position: Vector3
    velocity: Vector3
    boundingBox: AABB  // Axis-Aligned Bounding Box
    collisionMask: BitMask
    isStatic: Boolean
    onCollision: Function
END STRUCTURE

// Система виявлення зіткнень
CLASS CollisionDetectionSystem:
    staticRTree: RTree    // Для статичних об'єктів
    dynamicRTree: RTree   // Для динамічних об'єктів

    FUNCTION Initialize(gameObjects):
        FOR EACH obj IN gameObjects:
            IF obj.isStatic:
                staticRTree.insert(obj.boundingBox, obj)
            ELSE:
                dynamicRTree.insert(obj.boundingBox, obj)
            END IF
        END FOR
    END FUNCTION

    FUNCTION UpdateDynamicObject(obj, deltaTime):
```

```
        // Видаляємо старе положення
        dynamicRTree.delete(obj.boundingBox, obj)

        // Оновлюємо положення
        obj.position += obj.velocity * deltaTime
        obj.boundingBox = CalculateAABB(obj)

        // Вставляємо нове положення
        dynamicRTree.insert(obj.boundingBox, obj)
    END FUNCTION

    FUNCTION CheckCollisions():
        collisions = []

        // Перевірка динамічних об'єктів зі статичними
        dynamicObjects = dynamicRTree.getAllObjects()

        FOR EACH dynObj IN dynamicObjects:
            // Знаходимо потенційні зіткнення зі статичними об'єктами
            potentialStatic = staticRTree.search(dynObj.boundingBox)

            FOR EACH staticObj IN potentialStatic:
                IF CanCollide(dynObj, staticObj) AND
                   DetailedCollisionCheck(dynObj, staticObj):
                    collisions.append({dynObj, staticObj})
                END IF
            END FOR

            // Знаходимо потенційні зіткнення з іншими динамічними об
'єктами
            potentialDynamic = dynamicRTree.search(dynObj.boundingBox
)

            FOR EACH otherDyn IN potentialDynamic:
                IF otherDyn != dynObj AND
                   CanCollide(dynObj, otherDyn) AND
                   DetailedCollisionCheck(dynObj, otherDyn):
                    // Уникаємо дублювання пар
                    IF dynObj.id < otherDyn.id:
                        collisions.append({dynObj, otherDyn})
                    END IF
                END IF
            END FOR
        END FOR
```

```
        RETURN collisions
    END FUNCTION

    FUNCTION CanCollide(obj1, obj2):
        // Перевірка масок зіткнень
        RETURN (obj1.collisionMask & obj2.collisionMask) != 0
    END FUNCTION

    FUNCTION DetailedCollisionCheck(obj1, obj2):
        // Тут може бути більш точна перевірка (SAT, GJK алгоритми)
        RETURN TRUE  // Спрощено для прикладу
    END FUNCTION

    FUNCTION GetObjectsInViewFrustum(camera):
        // Отримання видимих об'єктів для рендерингу
        frustumBox = CalculateFrustumBoundingBox(camera)

        visibleStatic = staticRTree.search(frustumBox)
        visibleDynamic = dynamicRTree.search(frustumBox)

        // Додаткова перевірка на входження в піраміду видимості
        result = []

        FOR EACH obj IN visibleStatic + visibleDynamic:
            IF IsInFrustum(obj.boundingBox, camera):
                result.append(obj)
            END IF
        END FOR

        RETURN result
    END FUNCTION
END CLASS

// Ігровий цикл
collisionSystem = NEW CollisionDetectionSystem()
collisionSystem.Initialize(allGameObjects)

WHILE gameRunning:
    deltaTime = GetDeltaTime()

    // Оновлення позицій
    FOR EACH obj IN dynamicObjects:
        collisionSystem.UpdateDynamicObject(obj, deltaTime)
    END FOR
```

```
    // Перевірка зіткнень
    collisions = collisionSystem.CheckCollisions()

    // Обробка зіткнень
    FOR EACH collision IN collisions:
        collision.obj1.onCollision(collision.obj2)
        collision.obj2.onCollision(collision.obj1)
    END FOR

    // Рендеринг видимих об'єктів
    visibleObjects = collisionSystem.GetObjectsInViewFrustum(camera)
    RenderObjects(visibleObjects)
END WHILE
```

Приклад 3: Геопросторовий пошук в базі даних нерухомості

```
// Система пошуку нерухомості з геопросторовими запитами
CLASS RealEstateSearchSystem:
    propertyRTree: RTree
    properties: HashMap<Integer, Property>

    STRUCTURE Property:
        id: Integer
        location: Point
        boundary: Polygon
        boundingBox: Rectangle
        type: String  // apartment, house, land
        price: Float
        area: Float
        rooms: Integer
        amenities: List<String>
    END STRUCTURE

    FUNCTION SearchPropertiesNearLocation(center, radius, filters):
        // Створюємо область пошуку
        searchBox = Rectangle(
            center.x - radius,
            center.y - radius,
            center.x + radius,
            center.y + radius
        )

        // Пошук в R-дереві
        candidateIds = propertyRTree.search(searchBox)
        results = []
```

```
        FOR EACH id IN candidateIds:
            property = properties[id]

            // Точна перевірка відстані
            distance = CalculateDistance(center, property.location)

            IF distance <= radius AND MatchesFilters(property, filter
s):
                results.append({
                    property: property,
                    distance: distance
                })
            END IF
        END FOR

        // Сортування за відстанню
        results.sort((a, b) => a.distance - b.distance)

        RETURN results
    END FUNCTION

    FUNCTION FindPropertiesNearAmenities(amenityTypes, maxDistance):
        // Знаходимо всі необхідні зручності
        amenities = []
        FOR EACH type IN amenityTypes:
            amenities.extend(FindAmenitiesByType(type))
        END FOR

        // Для кожної нерухомості перевіряємо близькість до зручносте
й
        qualifiedProperties = []

        FOR EACH property IN properties.values():
            hasAllAmenities = TRUE

            FOR EACH amenityType IN amenityTypes:
                // Шукаємо найближчу зручність цього типу
                searchArea = ExpandingSearch(property.location, maxDi
stance)
                nearbyAmenities = amenityRTree.search(searchArea)

                foundNearby = FALSE
                FOR EACH amenity IN nearbyAmenities:
                    IF amenity.type == amenityType:
                        distance = CalculateDistance(property.locatio
```

```
n, amenity.location)
                        IF distance <= maxDistance:
                            foundNearby = TRUE
                            BREAK
                        END IF
                    END IF
                END FOR

                IF NOT foundNearby:
                    hasAllAmenities = FALSE
                    BREAK
                END IF
            END FOR

            IF hasAllAmenities:
                qualifiedProperties.append(property)
            END IF
        END FOR

        RETURN qualifiedProperties
    END FUNCTION

    FUNCTION AnalyzePropertyDensity(region):
        // Розбиваємо регіон на сітку для аналізу щільності
        grid = CreateGrid(region, GRID_CELL_SIZE)

        // Підраховуємо нерухомість у кожній клітинці
        FOR EACH cell IN grid:
            cellBounds = GetCellBounds(cell)
            propertiesInCell = propertyRTree.search(cellBounds)

            cell.count = propertiesInCell.length
            cell.totalValue = 0
            cell.averagePrice = 0

            FOR EACH propId IN propertiesInCell:
                property = properties[propId]
                cell.totalValue += property.price
            END FOR

            IF cell.count > 0:
                cell.averagePrice = cell.totalValue / cell.count
            END IF
        END FOR
```

```
        RETURN grid
    END FUNCTION
END CLASS

// Використання системи
searchSystem = NEW RealEstateSearchSystem()

// Пошук квартир поблизу метро
metroStation = Point(50.4501, 30.5234)  // Київ, станція метро
filters = {
    type: "apartment",
    minRooms: 2,
    maxPrice: 100000
}

nearbyProperties = searchSystem.SearchPropertiesNearLocation(
    metroStation,
    1000,  // 1 км
    filters
)

// Пошук будинків поблизу шкіл та парків
amenities = ["school", "park"]
houses = searchSystem.FindPropertiesNearAmenities(amenities, 500)  //
500 м

PRINT "Знайдено " + houses.length + " будинків поблизу шкіл та парків
"
```

Переваги та недоліки

Переваги R-дерева:

- Ефективність для просторових об’єктів — на відміну від KD-дерева, R-дерево спеціально розроблене для роботи з об’єктами, що мають геометричну протяжність (прямокутники, полігони)

- Збалансованість — R-дерево завжди залишається збалансованим, всі листи знаходяться на одному рівні, що гарантує логарифмічну складність операцій

- Гарна локальність даних — об’єкти, розташовані близько в просторі, зазвичай зберігаються в одних і тих самих вузлах, що покращує кешування

- Підтримка динамічних даних — ефективні операції вставки та видалення без необхідності повної перебудови структури

- Масштабованість — добре працює з великими обсягами даних, особливо при зберіганні на диску

- Універсальність — легко розширюється для роботи з різними типами просторових даних та запитів

Недоліки R-дерева:

- Перекриття MBR — мінімальні обмежуючі прямокутники можуть перекриватися, що призводить до необхідності перевіряти декілька гілок при пошуку

- Складність вибору оптимального розщеплення — якість дерева сильно залежить від алгоритму розщеплення вузлів

- Деградація при певних розподілах даних — для сильно кластеризованих або лінійно розподілених даних ефективність може знижуватися

- Складність реалізації — порівняно з простішими структурами, R-дерево вимагає більш складної логіки для підтримки інваріантів

- Неоптимальність для точкових даних — для роботи виключно з точками KD-дерево або квадродерево можуть бути ефективнішими

- Проблема “мертвого простору” — MBR можуть включати значні порожні області, особливо для об’єктів складної форми

Контрольні запитання

1. Що таке R-дерево і чим воно відрізняється від KD-дерева? Які основні властивості R-дерева?

2. Поясніть поняття мінімального обмежуючого прямокутника (MBR). Як він використовується в R-дереві?

3. Опишіть алгоритм пошуку об’єктів у заданій області за допомогою R-дерева. Яка його часова складність?

4. Як працює алгоритм вставки нового об’єкта в R-дерево? Що відбувається при переповненні вузла?

5. Поясніть квадратичний алгоритм розщеплення вузла. Які існують альтернативні методи?

6. Чому в R-дереві допускається перекриття MBR і як це впливає на продуктивність пошуку?

7. Як працює алгоритм видалення з R-дерева? Що таке конденсація дерева?

8. Які модифікації R-дерева існують (R*-дерево, R+-дерево) і в чому їх особливості?

9. Для яких типів просторових даних R-дерево найбільш ефективне? Наведіть приклади.

10. Як вибрати оптимальні параметри M (максимальна кількість записів) та m (мінімальна кількість записів) для R-дерева?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму R-дерева для індексування просторових об’єктів

2. Мета

Вивчити принципи роботи R-дерева, реалізувати основні операції (вставка, пошук, видалення), дослідити ефективність різних алгоритмів розщеплення вузлів та вплив параметрів на продуктивність структури.

3. Завдання

- Реалізувати класи Rectangle, RTreeEntry, RTreeNode

- Реалізувати алгоритм вставки з квадратичним розщепленням

- Реалізувати пошук об’єктів у заданій прямокутній області

- Реалізувати алгоритм видалення з конденсацією дерева

- Додати альтернативний метод розщеплення (лінійний або R*-tree split)

- Створити візуалізацію R-дерева для двовимірного випадку

- Провести експериментальне порівняння різних методів розщеплення

4. Опис

Лабораторна робота передбачає поетапну реалізацію R-дерева, починаючи з базових структур даних для представлення прямокутників та вузлів дерева. Студенти мають реалізувати різні варіанти алгоритмів розщеплення та експериментально дослідити їх вплив на якість індексу. Особлива увага приділяється візуалізації структури дерева та процесу виконання операцій.

5. Порядок виконання

- Створити клас Rectangle з методами для обчислення площі, перетину, об’єднання

- Реалізувати структури RTreeEntry та RTreeNode
- Створити клас RTree з базовими параметрами M та m
- Реалізувати метод chooseLeaf для вибору листа при вставці
- Реалізувати квадратичний алгоритм розщеплення (quadraticSplit)
- Додати метод adjustTree для коригування дерева після вставки
- Реалізувати метод search для пошуку в прямокутній області
- Додати метод delete з алгоритмом конденсації
- Реалізувати лінійний алгоритм розщеплення для порівняння
- Створити візуалізацію з відображенням MBR всіх рівнів
- Провести тестування на різних наборах даних
- Виміряти та порівняти продуктивність різних конфігурацій

6. Результат

Повнофункціональна реалізація R-дерева з підтримкою основних операцій, візуалізацією структури та результатами експериментального дослідження. Графіки залежності глибини дерева, середнього заповнення вузлів та часу виконання операцій від кількості об’єктів та параметрів дерева.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас Rectangle з методами area(), intersects(), contains(), combine(), enlargement()
- Класи RTreeEntry та RTreeNode з відповідними полями
- Клас RTree з методами insert(), search(), delete()
- Реалізація мінімум двох алгоритмів розщеплення
- Модуль візуалізації для відображення структури дерева

2. Продемонструвати роботу додатку:

- Побудова R-дерева для набору прямокутників з візуалізацією
- Демонстрація процесу вставки з розщепленням вузла
- Пошук об’єктів у заданій області з підсвічуванням результатів
- Видалення об’єктів з демонстрацією конденсації дерева
- Порівняння різних алгоритмів розщеплення на однакових даних
- Аналіз заповнення вузлів та перекриття MBR

3. Бути готовим відповісти на питання:

- Як вибирається вузол для вставки нового об’єкта?

- Чому важливо мінімізувати перекриття MBR?

- Як працює алгоритм pickSeeds при квадратичному розщепленні?

- Коли виникає необхідність у конденсації дерева?

- Які переваги дає R*-дерево порівняно з базовим R-деревом?

Тести

1. R-дерево призначене для індексування:

1) Тільки точкових даних

2) Просторових об’єктів з геометричною протяжністю

3) Одновимірних даних

4) Текстових даних

2. Всі листові вузли R-дерева:

1) Можуть знаходитися на різних рівнях

2) Завжди знаходяться на одному рівні

3) Містять тільки один об’єкт

4) Не містять MBR

3. При вставці нового об’єкта вузол для вставки вибирається за критерієм:

1) Мінімального збільшення площі MBR

2) Максимальної кількості об’єктів

3) Випадково

4) Мінімальної глибини

4. Квадратичний алгоритм розщеплення вибирає початкові об’єкти за критерієм:

1) Мінімальної відстані між ними

2) Максимального “марнування” простору при об’єднанні

3) Однакової площі

4) Алфавітного порядку

5. MBR у R-дереві можуть:

1) Тільки дотикатися

2) Перекриватися

3) Бути лише вкладеними один в одного

4) Мати лише квадратну форму

Відповіді:

1-2),

2-2),

3-1),

4-2),

5-2).

Тести підвищеної складності

1. Яка основна відмінність R*-дерева від базового R-дерева?

1) Заборона перекриття MBR

2) Використання сферичних областей замість прямокутних

3) Врахування перекриття та периметра при виборі вузла та розщепленні

4) Використання B-дерева для зберігання

2. Часова складність пошуку в R-дереві в найгіршому випадку:

1) $O(log\ n)$

2) $O(n)$

3) $O(\sqrt{n})$

4) $O(n\ log\ n)$

3. R+-дерево відрізняється від R-дерева тим, що:

1) Використовує циліндричні області

2) Забороняє перекриття MBR на одному рівні

3) Має змінну кількість дочірніх вузлів

4) Зберігає об’єкти тільки у внутрішніх вузлах

4. При видаленні об’єкта з R-дерева може виникнути:

1) Розщеплення вузла

2) Конденсація (злиття) вузлів

3) Ротація піддерев

4) Зміна висоти тільки одного піддерева

5. Гільбертово R-дерево використовує:

1) Фрактальну криву для впорядкування об’єктів

2) Випадкове розміщення об’єктів

3) Сортування за площею

4) Генетичні алгоритми

6. Для 3D просторових даних MBR в R-дереві представляється:

1) Сферою

2) Паралелепіпедом

3) Тетраедром

4) Циліндром

7. Priority R-tree оптимізує:

1) Швидкість вставки

2) Використання пам’яті

3) Перекриття MBR для найбільш важливих запитів

4) Паралельну обробку

8. Bulk loading в R-дереві використовується для:

1) Динамічної вставки об’єктів

2) Ефективної початкової побудови дерева з великого набору даних

3) Видалення групи об’єктів

4) Зміни параметрів дерева

9. STR (Sort-Tile-Recursive) метод побудови R-дерева:

1) Сортує об’єкти за однією координатою

2) Використовує рекурсивне розбиття простору на плитки

3) Випадково розподіляє об’єкти

4) Будує дерево знизу вгору

10. Для оцінки якості R-дерева НЕ використовується метрика:

1) Середнє перекриття MBR

2) Середня заповненість вузлів

3) Висота дерева

4) Кількість простих чисел серед ідентифікаторів об’єктів

Відповіді:

1-3. R*-дерево використовує складніші критерії, включаючи мінімізацію перекриття та периметра MBR.

2-2. У найгіршому випадку може знадобитися перевірити всі вузли через перекриття MBR.

3-2. R+-дерево усуває перекриття ціною можливого дублювання об’єктів у різних гілках.

4-2. При видаленні вузол може стати недозаповненим, що вимагає конденсації.

5-1. Крива Гільберта забезпечує хороше просторове впорядкування для пакетного завантаження.

6-2. У 3D просторі MBR - це паралелепіпед, вирівняний за осями координат.

7-3. Пріоритетне R-дерево оптимізує структуру для конкретних типів запитів.

8-2. Bulk loading дозволяє побудувати більш якісне дерево, знаючи всі дані заздалегідь.

9-2. STR рекурсивно розбиває простір на прямокутні плитки для групування об’єктів.

10-4. Ідентифікатори об’єктів не впливають на якість просторового індексу.

Завдання для самостійної роботи

1. Реалізуйте R*-дерево з примусовим перевставленням при розщепленні. Порівняйте якість дерева з базовим R-деревом на різних розподілах даних.

2. Розробіть алгоритм пакетного завантаження для R-дерева методом STR (Sort-Tile-Recursive). Порівняйте якість дерева з динамічною побудовою.

3. Реалізуйте R+-дерево без перекриття MBR. Дослідить вплив дублювання об’єктів на продуктивність та використання пам’яті.

4. Створіть гібридну структуру, що використовує R-дерево для щільних областей та квадродерево для розріджених.

5. Розробіть паралельну версію R-дерева з підтримкою одночасного доступу декількох потоків для читання та запису.

6. Реалізуйте 3D R-дерево для індексування тривимірних об’єктів. Візуалізуйте структуру за допомогою WebGL.

7. Створіть адаптивне R-дерево, що автоматично налаштовує параметри M та m залежно від розподілу даних.

8. Розробіть Гільбертове R-дерево, використовуючи криву Гільберта для впорядкування об’єктів при вставці.

9. Реалізуйте персистентне R-дерево, що зберігає історію всіх змін та дозволяє виконувати запити для попередніх версій.

10. Створіть розподілене R-дерево для обробки просторових даних, що не поміщаються в пам’ять одного сервера.

Теми лабораторних робіт

1. Базова реалізація R-дерева з квадратичним алгоритмом розщеплення та візуалізацією для 2D об’єктів.

2. Порівняльний аналіз алгоритмів розщеплення: лінійний, квадратичний та експоненційний.

3. Реалізація R*-дерева з оптимізаціями вибору вузла та примусовим перевставленням при переповненні.

4. Розробка системи просторового пошуку для ГІС з використанням R-дерева та реальних географічних даних.

5. Створення системи виявлення зіткнень для 2D гри з використанням R-дерева для оптимізації.

6. Реалізація пакетного завантаження алгоритмів (STR, Hilbert) та порівняння з динамічною побудовою.

7. Розробка просторової бази даних з R-деревом як основним індексом та підтримкою SQL-подібних запитів.

8. Дослідження впливу параметрів R-дерева (M, m, fill factor) на продуктивність різних операцій.

9. Створення веб-додатку для інтерактивної роботи з R-деревом: додавання, пошук, видалення об’єктів.

10. Реалізація розширень R-дерева для спеціальних випадків: часові дані (RT-дерево), невизначені дані (UR-дерево).

Розділ 12: Чисельні та криптографічні алгоритми

У цифрову епоху, коли кожна онлайн-транзакція, повідомлення чи завантаження файлу спирається на складні математичні операції, розуміння чисельних та криптографічних алгоритмів стає критично важливим. Цей розділ відкриває двері до захоплюючого світу, де абстрактна математика перетворюється на конкретні інструменти, що забезпечують безпеку наших даних, ефективність обчислень та надійність комунікацій.

Кожного разу, коли ви:

- Входите на захищений веб-сайт (HTTPS)

- Здійснюєте онлайн-платіж

- Обмінюєтеся зашифрованими повідомленнями

- Використовуєте цифровий підпис

- Стискаєте або обробляєте мультимедійні файли

...за лаштунками працюють алгоритми, які ми розглянемо в цьому розділі.

Три стовпи сучасних обчислень

1. Теорія чисел: фундамент криптографії

Теорія чисел, яку Карл Фрідріх Гаус називав "царицею математики", довгий час вважалася суто теоретичною дисципліною без практичного застосування. Сьогодні вона є основою практично всіх криптографічних систем.

Ми почнемо з алгоритму Евкліда — елегантного методу знаходження найбільшого спільного дільника, який, попри свій поважний вік (понад 2000 років), залишається одним з найважливіших інструментів у сучасній криптографії. Далі розглянемо китайську теорему про залишки — потужний інструмент для роботи з модульною арифметикою, який дозволяє значно прискорити криптографічні обчислення.

Особливу увагу приділимо простим числам — будівельним блокам криптографії. Ви дізнаєтеся, як алгоритм Міллера-Рабіна дозволяє швидко перевіряти, чи є величезне число простим, та як решета Ератосфена і Аткіна ефективно знаходять всі прості числа в заданому діапазоні.

2. Криптографія: мистецтво захисту інформації

Від стародавніх шифрів Цезаря до сучасних квантово-стійких алгоритмів — криптографія пройшла довгий шлях. У цьому підрозділі ми зануримося в серце сучасної криптографії.

Розпочнемо з RSA — алгоритму, який революціонізував світ безпечних комунікацій, дозволивши двом людям обмінюватися секретними повідомленнями без попередньої зустрічі. Вивчимо протокол Діффі-Геллмана, який елегантно вирішує проблему обміну ключами через незахищений канал.

Далі перейдемо до криптографії на еліптичних кривих — сучасного підходу, який забезпечує той самий рівень безпеки при значно менших розмірах ключів. Розглянемо хеш-функції — криптографічні "відбитки пальців" даних, без яких неможливі цифрові підписи, блокчейн та багато інших технологій. Завершимо огляд симетричним шифруванням AES — робочою конячкою сучасної криптографії, яка захищає все, від WiFi-з'єднань до державних секретів.

3. Алгебра: оптимізація обчислень

Ефективність — ключ до практичності. У третьому підрозділі ми розглянемо алгоритми, які дозволяють виконувати складні математичні операції з вражаючою швидкістю.

Швидке піднесення до степеня перетворює операцію, яка наївно вимагала б мільйони множень, на процедуру з десятками операцій. Це критично важливо для RSA та інших криптосистем. Швидке перетворення Фур'є (FFT) — один з найважливіших алгоритмів XX століття — лежить в основі обробки сигналів, стиснення даних та навіть множення великих чисел.

Алгоритм Штрассена показує, як нестандартне мислення може прискорити навіть такі базові операції, як множення матриць. А метод Гауса для розв'язання систем лінійних рівнянь залишається основним інструментом в чисельному аналізі, машинному навчанні та комп'ютерній графіці.

12.1. Теорія чисел

12.1.1. Алгоритм Евкліда

Розглянемо один з найфундаментальніших алгоритмів в математиці та інформатиці - алгоритм Евкліда для знаходження найбільшого спільного дільника двох чисел.

Теоретичні відомості

Алгоритм Евкліда - це ефективний метод обчислення найбільшого спільного дільника (НСД) двох натуральних чисел.

Найбільший спільний дільник двох чисел a і b - це найбільше натуральне число, яке ділить без остачі як a, так і b.

Позначається НСД(a, b) або gcd(a, b) від англійського greatest common divisor.

Алгоритм базується на простому, але потужному спостереженні: найбільший спільний дільник двох чисел не змінюється, якщо від більшого числа відняти менше. Точніше, для будь-яких двох натуральних чисел a і b, де a > b, виконується рівність:

НСД(a, b) = НСД(b, a mod b)

де a mod b означає остачу від ділення a на b.

Це твердження можна довести таким чином.

Нехай d = НСД(a, b). Тоді d ділить як a, так і b. Оскільки a = bq + r, де q - частка, а r - остача від ділення a на b, то r = a - bq. Якщо d ділить a і b, то d також ділить r = a - bq. Отже, d є спільним дільником b і r.

Аналогічно можна показати, що будь-який спільний дільник b і r також ділить a, тому НСД(a, b) = НСД(b, r).

Процес продовжується рекурсивно, поки одне з чисел не стане рівним нулю. Коли b = 0, НСД(a, 0) = a, оскільки будь-яке число ділить 0, і найбільшим дільником a є саме a.

Розглянемо послідовність остач, що виникає при застосуванні алгоритму Евкліда. Нехай маємо два числа a_0 і a_1, де $a_0 \geqslant a_1$. Виконуємо послідовні ділення з остачею:

$a_0 = a_1q_1 + a_2$, де $0 \leqslant a_2 < a_1$

$a_1 = a_2q_2 + a_3$, де $0 \leqslant a_3 < a_2$

$a_2 = a_3q_3 + a_4$, де $0 \leqslant a_4 < a_3$

...

$a_{n-2} = a_{n-1}q_{n-1} + a_n$, де $0 \leqslant a_n < a_{n-1}$

$a_{n-1} = a_nq_n + 0$

Оскільки послідовність остач $a_1 > a_2 > a_3 > \ldots > a_n > 0$ строго спадає і складається з невід'ємних цілих чисел, процес завершиться за скінченну кількість кроків. Останній ненульовий залишок a_n і є НСД(a_0, a_1).

Варіанти алгоритму

Існує кілька варіантів реалізації алгоритму Евкліда. Класичний алгоритм використовує операцію ділення з остачею. Існує також алгоритм Евкліда з відніманням, де замість ділення використовується послідовне віднімання меншого числа від більшого. Хоча цей варіант простіший для розуміння, він менш ефективний для великих чисел.

Розглянемо детально обидва варіанти алгоритму.

Алгоритм Евкліда з діленням

Цей варіант використовує операцію знаходження остачі від ділення:

```
функція НСД(a, b):
    поки b ≠ 0:
        temp = b
        b = a mod b
        a = temp
    повернути a
```

Алгоритм Евкліда з відніманням

Цей варіант використовує лише операції порівняння та віднімання:

```
функція НСД_віднімання(a, b):
    поки a ≠ b:
        якщо a > b:
            a = a - b
        інакше:
            b = b - a
    повернути a
```

Розширений алгоритм Евкліда

Розширений алгоритм Евкліда не лише знаходить НСД(a, b), але й обчислює коефіцієнти x та y такі, що ax + by = НСД(a, b). Це рівняння називається лінійним діофантовим рівнянням або тотожністю Безу.

```
функція розширений_НСД(a, b):
    якщо b = 0:
        повернути (a, 1, 0)
    інакше:
        (d, x1, y1) = розширений_НСД(b, a mod b)
        x = y1
        y = x1 - (a div b) * y1
        повернути (d, x, y)
```

де a div b означає цілу частину від ділення a на b.

Складність алгоритму

Часова складність алгоритму Евкліда є O(log min(a, b)). Це можна довести, використовуючи теорему Ламе, яка стверджує, що кількість кроків алгоритму Евкліда для двох чисел не перевищує п'ятикратної кількості цифр меншого числа. Точніше, якщо $a > b \geqslant 1$ і $b < F_{n+1}$ (де F_n - n-те число Фібоначчі), то алгоритм Евкліда виконає не більше n кроків.

Розглянемо детальний приклад обчислення НСД(252, 105) за допомогою алгоритму Евкліда.

Крок 1:

Ділимо 252 на 105 $252 = 105 \times 2 + 42$

Тепер працюємо з парою (105, 42)

Крок 2:

Ділимо 105 на 42 $105 = 42 \times 2 + 21$

Тепер працюємо з парою (42, 21)

Крок 3:

Ділимо 42 на 21 $42 = 21 \times 2 + 0$

Остача дорівнює 0, тому алгоритм завершується

Отже, НСД(252, 105) = 21.

Перевірка: $252 = 21 \times 12$, $105 = 21 \times 5$. Дійсно, 21 є найбільшим спільним дільником, оскільки 12 і 5 взаємно прості.

Область застосування

Алгоритм Евкліда має широке застосування в різних галузях математики та інформатики:

- Криптографія: В алгоритмі RSA розширений алгоритм Евкліда використовується для знаходження мультиплікативної оберненої за модулем. Це необхідно для генерації пар ключів шифрування та дешифрування.

- Теорія чисел: Алгоритм використовується для перевірки взаємної простоти чисел, знаходження коефіцієнтів лінійних діофантових рівнянь, обчислення неперервних дробів.

- Комп'ютерна алгебра: В системах символьних обчислень алгоритм Евкліда узагальнюється для знаходження НСД поліномів, що важливо для спрощення алгебраїчних виразів.

- Обробка сигналів: В цифровій обробці сигналів алгоритм використовується для знаходження періодів сигналів та синхронізації.

- Комп'ютерна графіка: При растеризації ліній алгоритм Брезенхема, який базується на ідеях алгоритму Евкліда, використовується для ефективного малювання прямих ліній на растрових дисплеях.

Приклади застосування

Приклад 1: Спрощення дробів

Для спрощення дробу 168/252 потрібно знайти НСД чисельника і знаменника:

НСД(168, 252) = НСД(168, 84) = НСД(84, 0) = 84

Тому 168/252 = (168÷84)/(252÷84) = 2/3

Приклад 2: Розв'язання лінійного діофантового рівняння

Знайдемо цілі розв'язки рівняння 47x + 30y = 1.

Використовуємо розширений алгоритм Евкліда:

47 = 30 × 1 + 17 30 = 17 × 1 + 13 17 = 13 × 1 + 4 13 = 4 × 3 + 1 4 = 1 × 4 + 0

Оскільки НСД(47, 30) = 1, рівняння має розв'язки. Працюючи в зворотному напрямку:

1 = 13 - 4 × 3 1 = 13 - (17 - 13) × 3 = 13 × 4 - 17 × 3 1 = (30 - 17) × 4 - 17 × 3 = 30 × 4 - 17 × 7 1 = 30 × 4 - (47 - 30) × 7 = 30 × 11 - 47 × 7

Отже, x = -7, y = 11 є частковим розв'язком.

Приклад 3: Генерація взаємно простих чисел для RSA

В криптосистемі RSA потрібно вибрати число e, взаємно просте з φ(n) = (p-1)(q-1). Використовуючи алгоритм Евкліда, можна швидко перевірити, чи НСД(e, φ(n)) = 1.

Переваги та недоліки

Переваги

- Ефективність: Алгоритм має логарифмічну часову складність O(log min(a,b)), що робить його надзвичайно швидким навіть для дуже великих чисел.

- Простота реалізації: Алгоритм легко реалізувати як рекурсивно, так і ітеративно, потребує лише базових арифметичних операцій.

- Універсальність: Алгоритм можна узагальнити для роботи з поліномами, гаусовими цілими числами та іншими евклідовими областями.

- Мінімальні вимоги до пам’яті: Алгоритм використовує O(1) додаткової пам’яті для ітеративної версії та O(log min(a,b)) для рекурсивної через стек викликів.

- Теоретична основа: Алгоритм має глибоке математичне обґрунтування і тісно пов’язаний з фундаментальними властивостями цілих чисел.

Недоліки

- Обмеженість застосування: Алгоритм працює тільки для об’єктів, що утворюють евклідову область (числа, поліноми).

- Проблеми з дуже великими числами: Хоча алгоритм ефективний, для криптографічних застосувань з числами в тисячі біт можуть знадобитися оптимізовані реалізації.

- Відсутність паралелізму: Класичний алгоритм Евкліда важко паралелізувати, оскільки кожен крок залежить від попереднього.

Контрольні запитання

1. Що таке найбільший спільний дільник двох чисел і як його позначають?

2. На якому математичному принципі базується алгоритм Евкліда?

3. Яка часова складність алгоритму Евкліда і від чого вона залежить?

4. У чому різниця між класичним та розширеним алгоритмом Евкліда?

5. Як алгоритм Евкліда використовується в криптографії RSA?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму Евкліда для знаходження найбільшого спільного дільника

2. Мета

Вивчити принципи роботи алгоритму Евкліда, реалізувати різні варіанти алгоритму, дослідити їх ефективність та застосувати для розв’язання практичних задач.

3. Завдання

- Реалізувати класичний алгоритм Евкліда (рекурсивну та ітеративну версії)

- Реалізувати алгоритм Евкліда з відніманням

- Реалізувати розширений алгоритм Евкліда

- Створити візуалізацію роботи алгоритму

- Порівняти ефективність різних реалізацій

- Застосувати алгоритм для спрощення дробів та розв’язання діофантових рівнянь

4. Опис

Лабораторна робота складається з кількох етапів. Спочатку необхідно реалізувати базові версії алгоритму Евкліда: рекурсивну та ітеративну. Потім реалізувати варіант з відніманням для порівняння ефективності. Розширений алгоритм Евкліда повинен повертати не тільки НСД, але й коефіцієнти лінійного представлення. Візуалізація повинна показувати покроковий процес обчислення з проміжними результатами. Для дослідження ефективності необхідно провести вимірювання часу виконання для різних пар чисел та побудувати графіки залежності часу від розміру вхідних даних.

5. Порядок виконання

1. Створити модуль з реалізаціями алгоритмів Евкліда

2. Реалізувати функції для кожного варіанту алгоритму

3. Створити модуль тестування з набором тестових даних

4. Реалізувати візуалізацію покрокового виконання

5. Провести порівняльний аналіз ефективності

6. Реалізувати застосування алгоритму для практичних задач

7. Оформити звіт з результатами досліджень

6. Результат

Програма повинна надавати користувачу можливість вибору методу обчислення НСД, вводу довільних чисел, перегляду покрокового виконання алгоритму. Додатково програма повинна демонструвати застосування алгоритму для спрощення дробів, розв’язання діофантових рівнянь та перевірки взаємної простоти чисел. Результати порівняння ефективності повинні бути представлені у вигляді таблиць та графіків.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функція класичного алгоритму Евкліда (рекурсивна версія)

- Функція класичного алгоритму Евкліда (ітеративна версія)
- Функція алгоритму Евкліда з відніманням
- Функція розширеного алгоритму Евкліда
- Модуль візуалізації покрокового виконання
- Модуль порівняння ефективності алгоритмів

2. Продемонструвати роботу додатку:
- Обчислення НСД для різних пар чисел
- Візуалізацію покрокового виконання алгоритму
- Порівняння часу виконання різних реалізацій
- Застосування для спрощення дробів
- Розв’язання лінійного діофантового рівняння

3. Бути готовим відповісти на питання:
- Чому алгоритм Евкліда завжди завершується?
- Яка максимальна кількість кроків алгоритму для заданої пари чисел?
- Як розширений алгоритм Евкліда знаходить коефіцієнти Безу?
- В яких випадках алгоритм з відніманням може бути неефективним?
- Як узагальнити алгоритм Евкліда для поліномів?

Тести

1. Алгоритм Евкліда базується на властивості:

1) НСД(a, b) = НСД(b, a mod b)

2) НСД(a, b) = НСД(a-b, b)

3) НСД(a, b) = НСД(a+b, b)

4) НСД(a, b) = НСД(a×b, b)

2. Часова складність алгоритму Евкліда:

1) O(n)

2) $O(n^2)$

3) O(log min(a,b))

4) O(max(a,b))

3. Розширений алгоритм Евкліда знаходить:

1) Тільки НСД двох чисел

2) НСД та НСК двох чисел

3) НСД та коефіцієнти x, y такі, що ax + by = НСД(a,b)

4) Всі дільники двох чисел

4. Для чисел 1071 та 462 алгоритм Евкліда дає НСД:

1) 21

2) 42

3) 7

4) 3

5. Алгоритм Евкліда завершується, коли:

1) Одне з чисел стає рівним 1

2) Остача від ділення дорівнює 0

3) Числа стають рівними

4) Кількість кроків перевищує log n

Відповіді:

1-1,

2-3,

3-3,

4-1,

5-2

Тести підвищеної складності

1. Які числа Фібоначчі дають найгірший випадок для алгоритму Евкліда?

1) F(n) та F(n+1)

2) F(n) та F(n-1)

3) F(n) та F(n+2)

4) F(n) та F(2n)

2. Для поліномів $f(x) = x^3 + 2x^2 + x$ та $g(x) = x^2 + x$ алгоритм Евкліда дає:

1) x

2) x + 1

3) $x^2 + x$

4) 1

3. Складність розширеного алгоритму Евкліда порівняно з класичним:

1) Однакова асимптотично

2) Вища на логарифмічний множник

3) Вища на лінійний множник

4) Квадратична замість логарифмічної

4. Для знаходження мультиплікативної оберненої числа a за модулем m необхідно:

1) НСД(a, m) = 1 та використати розширений алгоритм Евкліда

2) НСД(a, m) = m

3) a < m

4) a та m - прості числа

5. Бінарний алгоритм Евкліда (алгоритм Стейна) оптимізований для:

1) Великих простих чисел

2) Двійкового представлення чисел на комп’ютері

3) Чисел Фібоначчі

4) Від’ємних чисел

6. У випадку НСД(F(n), F(n-1)), де F(n) - n-те число Фібоначчі, результат завжди:

1) F(n-2)

2) 1

3) F(1) = 1

4) n

7. Для трьох чисел a, b, c правильна формула обчислення НСД:

1) НСД(a, b, c) = НСД(НСД(a, b), c)

2) НСД(a, b, c) = НСД(a, b) × НСД(b, c)

3) НСД(a, b, c) = НСД(a + b + c, abc)

4) НСД(a, b, c) = НСД(a, b) + НСД(b, c) - b

8. Лема Безу стверджує, що для НСД(a, b) = d:

1) d ділить будь-яку лінійну комбінацію a та b

2) існують цілі x, y такі, що ax + by = d

3) d = 1 тоді і тільки тоді, коли a та b взаємно прості

4) всі варіанти вірні

9. При обчисленні НСД(2^n - 1, 2^m - 1) результат:

1) 2^НСД(n,m) - 1

2) 2^(n+m) - 1

3) 1

4) 2^min(n,m) - 1

10. Алгоритм Евкліда для гаусових цілих чисел (a + bi):

1) Не існує

2) Використовує норму $N(a + bi) = a^2 + b^2$

3) Працює тільки для дійсної частини

4) Вимагає комплексного ділення

Відповіді:

1-2 Послідовні числа Фібоначчі дають максимальну кількість кроків алгоритму

2-3 Спільний дільник x(x+1) виноситься за дужки

3-1 Обидва алгоритми мають складність O(log min(a,b))

4-1 Обернений елемент існує тоді і тільки тоді, коли числа взаємно прості

5-2 Використовує операції зсуву замість ділення для ефективності

6-2 Послідовні числа Фібоначчі завжди взаємно прості

7-1 НСД кількох чисел обчислюється послідовно

8-4 Всі твердження є наслідками леми Безу

9-1 Це властивість чисел Мерсенна

10-2 Алгоритм використовує евклідову норму для впорядкування

Завдання для самостійної роботи

1. Реалізуйте бінарний алгоритм Евкліда (алгоритм Стейна), який використовує лише операції зсуву, порівняння та віднімання. Порівняйте його ефективність з класичним алгоритмом.

2. Напишіть програму для знаходження НСД трьох і більше чисел. Дослідить різні підходи до обчислення НСД(a_1, a_2, …, a_n).

3. Реалізуйте алгоритм Евкліда для поліномів з цілими коефіцієнтами. Продемонструйте його роботу на конкретних прикладах.

4. Створіть візуалізацію геометричної інтерпретації алгоритму Евкліда як процесу покриття прямокутника квадратами.

5. Напишіть програму для знаходження всіх розв'язків лінійного діофантового рівняння ax + by = c, використовуючи розширений алгоритм Евкліда.

6. Дослідіть зв'язок між алгоритмом Евкліда та неперервними дробами. Реалізуйте алгоритм розкладу раціонального числа в неперервний дріб.

7. Реалізуйте паралельну версію алгоритму Евкліда для кількох пар чисел одночасно. Виміряйте прискорення на багатоядерному процесорі.

8. Створіть програму для генерації пар взаємно простих чисел заданого розміру для криптографічних застосувань.

9. Напишіть програму, яка знаходить найменше спільне кратне (НСК) двох чисел, використовуючи формулу НСК(a,b) = |a×b|/НСД(a,b).

10. Реалізуйте узагальнений алгоритм Евкліда для гаусових цілих чисел (комплексних чисел виду a + bi, де a та b - цілі).

Теми лабораторних робіт

1. Базова реалізація алгоритму Евкліда та його застосування для спрощення дробів

2. Розширений алгоритм Евкліда та розв'язання лінійних діофантових рівнянь

3. Порівняльний аналіз ефективності різних варіантів алгоритму Евкліда

4. Застосування алгоритму Евкліда в модулярній арифметиці та криптографії

5. Алгоритм Евкліда для поліномів та його застосування в комп'ютерній алгебрі

6. Візуалізація алгоритму Евкліда: створення інтерактивної навчальної програми

7. Бінарний алгоритм Евкліда та оптимізації для комп'ютерних обчислень

8. Паралельні та розподілені реалізації алгоритму Евкліда для великих чисел

9. Застосування алгоритму Евкліда для генерації RSA-ключів

10. Узагальнення алгоритму Евкліда: евклідові області та їх властивості

12.1.2. Китайська теорема про залишки

Теоретичні відомості

Китайська теорема про залишки є одним з найважливіших результатів теорії чисел, який дозволяє розв'язувати системи лінійних конгруенцій з попарно взаємно простими модулями. Ця теорема має глибоке історичне коріння і знаходить широке застосування в сучасній криптографії, комп'ютерних науках та математиці.

Формулювання теореми

Нехай m_1, m_2, ..., m_k - попарно взаємно прості натуральні числа, тобто НСД(m_i, m_j) = 1 для всіх i ≠ j. Тоді система конгруенцій:

$x \equiv a_1 \pmod{m_1}$ $x \equiv a_2 \pmod{m_2}$ ... $x \equiv a_k \pmod{m_k}$

має єдиний розв'язок за модулем $M = m_1 \cdot m_2 \cdot \ldots \cdot m_k$.

Це означає, що існує єдине ціле число x в інтервалі [0, M-1], яке задовольняє всі конгруенції одночасно. Більш того, всі розв'язки системи мають вигляд x + tM, де t - довільне ціле число.

Конструктивне доведення та алгоритм

Доведення теореми є конструктивним і надає явну формулу для знаходження розв'язку. Для кожного i від 1 до k визначимо:

$M_i = M / m_i = m_1 \cdot m_2 \cdot \ldots \cdot m_{i-1} \cdot m_{i+1} \cdot \ldots \cdot m_k$

Оскільки НСД(M_i, m_i) = 1 (бо M_i є добутком всіх модулів, крім m_i, а всі модулі попарно взаємно прості), то існує обернений елемент y_i такий, що:

$M_i \cdot y_i \equiv 1 \pmod{m_i}$

Тоді розв'язок системи конгруенцій дається формулою:

$x = (a_1 \cdot M_1 \cdot y_1 + a_2 \cdot M_2 \cdot y_2 + \ldots + a_k \cdot M_k \cdot y_k) \bmod M$

Наведемо алгоритм розв'язання.

Розглянемо покроковий алгоритм застосування китайської теореми про залишки:

```
функція китайська_теорема(a[], m[], k):
    // a[] - масив залишків a₁, a₂, ..., aₖ
    // m[] - масив модулів m₁, m₂, ..., mₖ
    // k - кількість конгруенцій
```

```
    // Крок 1: Перевірка попарної взаємної простоти
    для i від 0 до k-1:
        для j від i+1 до k-1:
            якщо НСД(m[i], m[j]) ≠ 1:
                повернути "Модулі не є попарно взаємно простими"

    // Крок 2: Обчислення загального модуля M
    M = 1
    для i від 0 до k-1:
        M = M * m[i]

    // Крок 3: Обчислення часткових добутків та обернених елементів
    x = 0
    для i від 0 до k-1:
        Mi = M / m[i]
        yi = обернений_елемент(Mi, m[i])
        x = x + a[i] * Mi * yi

    // Крок 4: Приведення результату за модулем M
    x = x mod M
    повернути x
```

Алгоритм знаходження оберненого елемента

Для знаходження оберненого елемента використовується розширений алгоритм Евкліда:

```
функція обернений_елемент(a, m):
    // Знаходить x такий, що a*x ≡ 1 (mod m)
    // Використовує розширений алгоритм Евкліда

    якщо НСД(a, m) ≠ 1:
        повернути "Обернений елемент не існує"

    // Ініціалізація
    old_r = a
    r = m
    old_s = 1
    s = 0

    поки r ≠ 0:
        quotient = old_r div r
        temp = r
        r = old_r - quotient * r
        old_r = temp
```

```
        temp = s
        s = old_s - quotient * s
        old_s = temp

    // old_s містить обернений елемент
    якщо old_s < 0:
        old_s = old_s + m

    повернути old_s
```

Розглянемо, чому запропонована формула дає правильний розв'язок. Для кожного доданка $a_i \cdot M_i \cdot y_i$ маємо:

- При $j \neq i$: $a_i \cdot M_i \cdot y_i \equiv 0 \pmod{m_j}$, оскільки M_i ділиться на m_j

- При $j = i$: $a_i \cdot M_i \cdot y_i \equiv a_i \cdot 1 \equiv a_i \pmod{m_i}$, оскільки $M_i \cdot y_i \equiv 1 \pmod{m_i}$

Таким чином, сума всіх доданків $x = \Sigma(a_i \cdot M_i \cdot y_i)$ задовольняє всі конгруенції системи.

Альтернативний метод: послідовна підстановка

Існує альтернативний метод розв'язання системи конгруенцій, який не вимагає обчислення обернених елементів для великих чисел:

```
функція послідовна_підстановка(a[], m[], k):
    x = a[0]
    M = m[0]

    для i від 1 до k-1:
        // Розв'язуємо x ≡ a[i] (mod m[i])
        // з урахуванням x ≡ x_поточне (mod M)

        поки x mod m[i] ≠ a[i]:
            x = x + M

        M = M * m[i]

    повернути x
```

Оптимізований алгоритм Гарнера

Алгоритм Гарнера є ефективною модифікацією китайської теореми про залишки, особливо корисною коли потрібно працювати з дуже великими числами:

```
функція алгоритм_гарнера(a[], m[], k):
    // Попередні обчислення
    c[][] = новий_масив[k][k]
    для i від 0 до k-1:
```

```
        c[i][i] = 1
        для j від 0 до i-1:
            c[i][j] = обернений_елемент(m[j], m[i])
            для l від j+1 до i-1:
                c[i][j] = (c[i][j] * c[i][l]) mod m[i]

    // Основний алгоритм
    x[] = новий_масив[k]
    для i від 0 до k-1:
        x[i] = a[i]
        для j від 0 до i-1:
            x[i] = (x[i] - x[j]) * c[i][j] mod m[i]
            якщо x[i] < 0:
                x[i] = x[i] + m[i]

    // Відновлення результату
    результат = x[0]
    добуток = 1
    для i від 1 до k-1:
        добуток = добуток * m[i-1]
        результат = результат + x[i] * добуток

    повернути результат
```

Розв’яжемо систему конгруенцій:

- $x \equiv 2 \pmod 3$

- $x \equiv 3 \pmod 5$

- $x \equiv 2 \pmod 7$

Крок 1: Перевіряємо попарну взаємну простоту модулів:

- НСД(3, 5) = 1 ✓

- НСД(3, 7) = 1 ✓

- НСД(5, 7) = 1 ✓

Крок 2: Обчислюємо загальний модуль:

$M = 3 \times 5 \times 7 = 105$

Крок 3: Обчислюємо часткові добутки:

- $M_1 = M/3 = 105/3 = 35$

- $M_2 = M/5 = 105/5 = 21$

- $M_3 = M/7 = 105/7 = 15$

Крок 4: Знаходимо обернені елементи:

- Для $M_1 = 35$ за модулем 3: $35 \equiv 2 \pmod 3$, потрібно знайти y_1 таке, що $2y_1 \equiv 1 \pmod 3$. Отримуємо $y_1 = 2$, бо $2 \times 2 = 4 \equiv 1 \pmod 3$

- Для $M_2 = 21$ за модулем 5: $21 \equiv 1 \pmod 5$, потрібно знайти y_2 таке, що $1 \times y_2 \equiv 1 \pmod 5$. Отримуємо $y_2 = 1$

- Для $M_3 = 15$ за модулем 7: $15 \equiv 1 \pmod 7$, потрібно знайти y_3 таке, що $1 \times y_3 \equiv 1 \pmod 7$. Отримуємо $y_3 = 1$

Крок 5: Обчислюємо розв'язок:

$x = (2 \times 35 \times 2 + 3 \times 21 \times 1 + 2 \times 15 \times 1) \bmod 105$ $x = (140 + 63 + 30) \bmod 105$ $x = 233 \bmod 105$ $x = 23$

Перевірка:

- $23 \div 3 = 7$ залишок 2 ✓

- $23 \div 5 = 4$ залишок 3 ✓

- $23 \div 7 = 3$ залишок 2 ✓

Отже, $x = 23$ є розв'язком системи.

Область застосування

Китайська теорема про залишки має широке застосування в різних галузях математики та інформатики:

Криптографія

- RSA криптосистема: Використовується для прискорення операцій шифрування та дешифрування. Замість обчислення $x^d \bmod n$, де $n = p \times q$, обчислюються $x^d \bmod p$ та $x^d \bmod q$ окремо, а потім результат відновлюється за допомогою китайської теореми.

- Схеми розділення секрету: Дозволяє розділити секрет на частини так, що для його відновлення потрібні всі частини.

- Криптографічні протоколи: Використовується для ефективних обчислень в скінченних полях та кільцях.

Комп'ютерні науки

- Паралельні обчислення: Дозволяє розбити велику задачу на незалежні підзадачі з меншими модулями.

- Обробка сигналів: Використовується в алгоритмах швидкого перетворення Фур'є для оптимізації обчислень.

- Системи комп’ютерної алгебри: Для роботи з великими числами та поліномами.

Теорія чисел

- Розв’язання діофантових рівнянь: Допомагає знаходити цілочисельні розв’язки систем рівнянь.

- Теорія конгруенцій: Є основним інструментом для роботи з системами конгруенцій.

- Факторизація чисел: Використовується в деяких алгоритмах факторизації.

Приклади застосування

Приклад 1: Прискорення RSA

У криптосистемі RSA для дешифрування повідомлення c обчислюється $m = c^d \bmod n$, де $n = p \times q$. Використовуючи китайську теорему:

1. Обчислюємо $m_1 = c^d \bmod p$

2. Обчислюємо $m_2 = c^d \bmod q$

3. Відновлюємо m за допомогою китайської теореми

Це дає прискорення приблизно в 4 рази, оскільки операції з числами вдвічі меншого розміру виконуються вчетверо швидше.

Приклад 2: Календарні обчислення

Знайдемо рік, який дає залишок 1 при діленні на 4 (високосний наступного року), залишок 6 при діленні на 7 (субота 1 січня) та залишок 10 при діленні на 12 (рік Собаки):

Система конгруенцій: - $x \equiv 1 \pmod 4$ - $x \equiv 6 \pmod 7$ - $x \equiv 10 \pmod{12}$

Але НСД(4, 12) = 4 ≠ 1, тому спочатку потрібно перетворити систему. Оскільки $x \equiv 1 \pmod 4$ і $x \equiv 10 \pmod{12}$, перевіримо сумісність: $10 \equiv 2 \pmod 4 \neq 1$, тому система несумісна в початковому вигляді.

Приклад 3: Розподілені обчислення

Для обчислення $2^{1000} \bmod 1001$, де $1001 = 7 \times 11 \times 13$:

1. Обчислюємо $2^{1000} \bmod 7 = 2^4 = 16 \equiv 2 \pmod 7$

2. Обчислюємо $2^{1000} \bmod 11 = 2^{10} = 1024 \equiv 1 \pmod{11}$

3. Обчислюємо $2^{1000} \bmod 13 = 2^{12} = 4096 \equiv 1 \pmod{13}$

Застосовуючи китайську теорему для системи: - $x \equiv 2 \pmod 7$ - $x \equiv 1 \pmod{11}$ - $x \equiv 1 \pmod{13}$

Отримуємо x = 793.

Переваги та недоліки

Переваги

- Ефективність обчислень: Дозволяє замінити операції з великими числами на операції з меншими, що значно прискорює обчислення.

- Паралелізація: Обчислення за різними модулями можна виконувати незалежно і паралельно.

- Точність: Дає точний результат без похибок округлення, на відміну від методів з плаваючою комою.

- Універсальність: Може бути узагальнена для кілець головних ідеалів та інших алгебраїчних структур.

- Простота реалізації: Базовий алгоритм легко реалізувати, використовуючи лише арифметичні операції.

Недоліки

- Обмеження на модулі: Вимагає попарної взаємної простоти модулів, що не завжди виконується на практиці.

- Зростання проміжних результатів: При обчисленні можуть виникати дуже великі проміжні значення.

- Складність порівняння: Важко порівнювати числа, представлені у вигляді залишків за різними модулями.

- Обмежений діапазон: Результат визначений тільки за модулем добутку всіх модулів.

- Чутливість до помилок: Помилка в одному залишку призводить до повністю неправильного результату.

Контрольні запитання

1. Яка основна умова застосовності китайської теореми про залишки?

2. Як обчислюється загальний модуль M у китайській теоремі про залишки?

3. Що таке часткові добутки M_i і як вони використовуються в алгоритмі?

4. Чому для кожного часткового добутку M_i існує обернений елемент за модулем m_i?

5. Як китайська теорема про залишки використовується для прискорення операцій в криптосистемі RSA?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження китайської теореми про залишки

2. Мета

Вивчити китайську теорему про залишки, реалізувати алгоритми її застосування, дослідити ефективність при розв'язанні практичних задач та порівняти різні методи реалізації.

3. Завдання

- Реалізувати базовий алгоритм китайської теореми про залишки
- Реалізувати алгоритм Гарнера для роботи з великими числами
- Створити функцію перевірки сумісності системи конгруенцій
- Реалізувати застосування для прискорення RSA операцій
- Провести порівняльний аналіз різних методів
- Створити візуалізацію процесу розв'язання

4. Опис

Лабораторна робота включає реалізацію різних варіантів алгоритму китайської теореми про залишки. Необхідно створити модуль для роботи з системами конгруенцій, який включає перевірку умов застосовності теореми, базовий алгоритм розв'язання, оптимізований алгоритм Гарнера. Окремо слід реалізувати застосування теореми для криптографічних обчислень, зокрема прискорення операцій RSA. Програма повинна включати інтерфейс для введення систем конгруенцій, візуалізацію кроків розв'язання та порівняння продуктивності різних методів.

5. Порядок виконання

1. Створити модуль для обчислення НСД та обернених елементів
2. Реалізувати функцію перевірки попарної взаємної простоти
3. Реалізувати базовий алгоритм китайської теореми
4. Реалізувати алгоритм Гарнера
5. Створити модуль для RSA операцій з використанням китайської теореми
6. Реалізувати візуалізацію процесу розв'язання
7. Провести тестування на різних наборах даних
8. Виконати порівняльний аналіз продуктивності

6. Результат

Програма повинна надавати можливість розв’язувати системи лінійних конгруенцій різними методами, автоматично перевіряти умови застосовності китайської теореми, візуалізувати процес розв’язання з детальним поясненням кожного кроку. Для криптографічних застосувань програма повинна демонструвати прискорення RSA операцій при використанні китайської теореми порівняно зі звичайним методом. Результати порівняння різних алгоритмів повинні бути представлені у вигляді графіків та таблиць.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Модуль обчислення НСД та обернених елементів
- Функція перевірки попарної взаємної простоти модулів
- Реалізація базового алгоритму китайської теореми
- Реалізація алгоритму Гарнера
- Модуль прискорення RSA операцій
- Модуль візуалізації розв’язання

2. Продемонструвати роботу додатку:

- Розв’язання систем конгруенцій з різною кількістю рівнянь
- Обробка випадків несумісних систем
- Порівняння швидкості базового алгоритму та алгоритму Гарнера
- Демонстрація прискорення RSA операцій
- Візуалізація покрокового розв’язання

3. Бути готовим відповісти на питання:

- Чому модулі повинні бути попарно взаємно простими?
- Як працює алгоритм знаходження оберненого елемента?
- У чому переваги алгоритму Гарнера над базовим алгоритмом?
- Як перевірити правильність розв’язку системи конгруенцій?
- Яке прискорення дає використання китайської теореми в RSA?

Тести

1. Китайська теорема про залишки застосовна, коли модулі є:

1) Попарно взаємно простими

2) Простими числами

3) Степенями двійки

4) Послідовними числами

2. Для системи $x \equiv 2 \pmod 3$, $x \equiv 3 \pmod 4$, $x \equiv 1 \pmod 5$ розв’язок за модулем 60 дорівнює:

1) 11

2) 23

3) 41

4) 51

3. У формулі розв’язку $x = \Sigma(a_i M_i y_i) \bmod M$, величина M_i дорівнює:

1) M / m_i

2) $M \times m_i$

3) m_i / M

4) НСД(M, m_i)

4. Алгоритм Гарнера відрізняється від базового алгоритму:

1) Не потребує обчислення великих проміжних результатів

2) Працює тільки для простих модулів

3) Має експоненційну складність

4) Не гарантує правильність результату

5. При використанні китайської теореми в RSA для $n = p \times q$ обчислення прискорюються приблизно в:

1) 2 рази

2) 4 рази

3) p разів

4) $p \times q$ разів

Відповіді:

1-1,

2-1,

3-1,

4-1,

5-2

Тести підвищеної складності

1. Для якої з наведених систем конгруенцій НЕ можна застосувати китайську теорему безпосередньо?

1) $x \equiv 1 \pmod 4$, $x \equiv 2 \pmod 9$, $x \equiv 3 \pmod{25}$

2) $x \equiv 2 \pmod 6$, $x \equiv 3 \pmod{10}$, $x \equiv 4 \pmod{15}$

3) $x \equiv 1 \pmod 7$, $x \equiv 2 \pmod{11}$, $x \equiv 3 \pmod{13}$

4) $x \equiv 0 \pmod 3$, $x \equiv 0 \pmod 5$, $x \equiv 0 \pmod 7$

2. Якщо система $x \equiv a \pmod m$, $x \equiv b \pmod n$ має розв'язок при НСД(m,n) = d > 1, то необхідною умовою є:

1) $a \equiv b \pmod d$

2) $a \equiv b \pmod{mn}$

3) d ділить a та b

4) m ділить b - a

3. Складність базового алгоритму китайської теореми для k конгруенцій з модулями до n:

1) $O(k \log n)$

2) $O(k^2 \log n)$

3) $O(k \log^2 n)$

4) $O(k^2 n)$

4. При представленні числа X у системі залишкових класів з модулями $m_1, \ldots, m_k$, операція додавання:

1) Виконується покомпонентно за $O(k)$ операцій

2) Вимагає повного відновлення числа

3) Має складність $O(k^2)$

4) Неможлива в цьому представленні

5. Для розширення китайської теореми на випадок НСД(m_i, m_j) = $d_{ij} > 1$ необхідно:

1) Перевірити сумісність $a_i \equiv a_j \pmod{d_{ij}}$ для всіх пар

2) Замінити всі модулі на їх НСК

3) Використати лише прості модулі

4) Застосувати алгоритм Евкліда

6. У схемі розділення секрету Асмута-Блума на основі китайської теореми:

1) Секрет можна відновити з будь-якої підмножини достатнього розміру

2) Потрібні всі частини для відновлення

3) Використовуються тільки прості модулі

4) Секрет зберігається в явному вигляді

7. При факторизації числа $n = p \times q$ за допомогою китайської теореми, знаючи $x^2 \equiv y^2 \pmod{n}$, але $x \not\equiv \pm y \pmod{n}$:

1) НСД(x-y, n) дає нетривіальний дільник n

2) НСД(x+y, n) = n завжди

3) Факторизація неможлива

4) Потрібно знати $\varphi(n)$

8. Для системи $x \equiv a_i \pmod{p_i^{k_i}}$, де p_i - різні прості числа, узагальнена китайська теорема:

1) Застосовна безпосередньо

2) Не застосовна

3) Вимагає модифікації для однакових p

4) Працює тільки при k = 1

9. При обчисленні дискретного логарифма методом Поліга-Хеллмана використовується китайська теорема для:

1) Зведення задачі до підгруп простого порядку

2) Прискорення множення

3) Факторизації модуля

4) Генерації випадкових чисел

10. Квадратична складність появляється при використанні китайської теореми для:

1) Множення чисел у системі залишкових класів

2) Додавання чисел

3) Порівняння чисел

4) Ділення з остачею

Відповіді:

1-2 НСД(6,10)=2, НСД(6,15)=3, НСД(10,15)=5 - модулі не попарно взаємно прості

2-1 Для сумісності системи необхідно, щоб залишки були конгруентні за модулем НСД

3-3 Для кожної конгруенції потрібно знайти обернений елемент - O(log n), всього k конгруенцій, плюс k множень великих чисел

4-1 Додавання в системі залишкових класів виконується незалежно для кожного модуля

5-1 Узагальнена китайська теорема вимагає перевірки умов сумісності для всіх пар модулів

6-1 Схема Асмута-Блума дозволяє відновити секрет з k з n частин, де k - поріг

7-1 Якщо $x^2 \equiv y^2 \pmod n$, то n ділить (x-y)(x+y), і НСД дає дільник

8-1 Степені різних простих чисел попарно взаємно прості

9-1 Метод Поліга-Хеллмана розкладає порядок групи і розв’язує задачу в підгрупах

10-3 Порівняння вимагає відновлення чисел з системи залишків або складних обчислень

Завдання для самостійної роботи

1. Реалізуйте узагальнену китайську теорему для випадку, коли модулі не є попарно взаємно простими. Алгоритм повинен перевіряти сумісність системи та знаходити розв’язок, якщо він існує.

2. Створіть програму для арифметики в системі залишкових класів: реалізуйте додавання, віднімання, множення та порівняння чисел, представлених як набори залишків.

3. Реалізуйте схему розділення секрету Асмута-Блума, яка базується на китайській теоремі про залишки. Програма повинна розділяти секрет на n частин так, щоб будь-які k частин дозволяли його відновити.

4. Напишіть програму для прискорення обчислення факторіалу великих чисел за допомогою китайської теореми. Порівняйте швидкість з прямим обчисленням.

5. Реалізуйте алгоритм розв’язання систем поліноміальних конгруенцій $f(x) \equiv 0 \pmod{m_i}$ для попарно взаємно простих модулів.

6. Створіть візуалізацію геометричної інтерпретації китайської теореми як ізоморфізму між Z_n та прямим добутком Z_{mi}.

7. Реалізуйте ефективний алгоритм переведення числа з десяткової системи в систему залишкових класів та навпаки для заданого набору модулів.

8. Напишіть програму для знаходження всіх систем конгруенцій з трьома модулями до 20, які мають розв’язок $x = 100$.

9. Реалізуйте паралельну версію алгоритму китайської теореми, де обчислення для різних модулів виконуються в окремих потоках.

10. Створіть програму для дослідження розподілу розв’язків випадкових систем конгруенцій. Проаналізуйте статистичні властивості розв’язків.

Теми лабораторних робіт

1. Базова реалізація китайської теореми про залишки та її застосування для розв’язання систем конгруенцій

2. Арифметика в системі залишкових класів: реалізація основних операцій та порівняння ефективності

3. Прискорення криптографічних операцій RSA за допомогою китайської теореми

4. Реалізація та дослідження алгоритму Гарнера для роботи з великими числами

5. Схеми розділення секрету на основі китайської теореми про залишки

6. Узагальнена китайська теорема для систем з непопарно взаємно простими модулями

7. Застосування китайської теореми в алгоритмах факторизації чисел

8. Паралельні алгоритми на основі китайської теореми про залишки

9. Китайська теорема для поліномів та її застосування в комп’ютерній алгебрі

10. Оптимізація обчислення дискретного перетворення Фур’є за допомогою китайської теореми

Тест Міллера-Рабіна на простоту

Теоретичні відомості

Тест Міллера-Рабіна є одним з найефективніших імовірнісних алгоритмів перевірки чисел на простоту. Цей алгоритм базується на властивостях простих чисел у модулярній арифметиці та дозволяє з високою ймовірністю визначити, чи є задане число простим або складеним.

Алгоритм Міллера-Рабіна базується на двох фундаментальних теоремах теорії чисел. Перша з них - це мала теорема Ферма, яка стверджує, що якщо p - просте число, а a - ціле число, не кратне p, то:

$$a^{(p-1)} \equiv 1 \, (mod \, p)$$

Ця теорема дає необхідну умову простоти числа. Якщо для деякого a виконується $a^{(n-1)} \not\equiv 1 \, (mod \, n)$, то n точно є складеним числом. Однак зворотне твердження не завжди вірне - існують складені числа, для яких ця умова виконується. Такі числа називаються псевдопростими за основою a.

Друга важлива властивість стосується квадратних коренів з одиниці за модулем простого числа. Якщо p - непарне просте число і $x^2 \equiv 1 \, (mod \, p)$, то $x \equiv \pm 1 \, (mod \, p)$. Це означає, що єдиними квадратними коренями з одиниці за модулем простого числа є 1 та -1 (або p-1).

Основна ідея тесту Міллера-Рабіна

Тест Міллера-Рабіна поєднує ці дві властивості для створення потужного критерію простоти. Для перевірки числа n на простоту, спочатку представляємо n-1 у вигляді:

$$n - 1 = 2^s \times d$$

де d - непарне число, а s - кількість разів, які 2 ділить n-1. Це представлення завжди можливе для непарного n.

Далі вибираємо випадкове число a з інтервалу [2, n-2] і обчислюємо послідовність:

$$a^d \, mod \, n, a^{2d} mod \, n, a^{4d} mod \, n, \dots, a^{(2(s-1)\times d)} \, mod \, n, a^{(2s)\times d)} \, mod \, n$$

Остання величина в цій послідовності дорівнює $a^{(n-1)} \, mod \, n$. Якщо n просте, то за малою теоремою Ферма це значення повинно дорівнювати 1. Крім того, кожен елемент послідовності є квадратом попереднього, тому якщо деякий елемент дорівнює 1, то попередній повинен дорівнювати ±1.

Критерій складеності

Число n є складеним, якщо для обраного свідка a виконується одна з умов:

- $a^d \not\equiv 1 \ (mod\ n)$ і $a^{(2r \times d)} \not\equiv -1 \ (mod\ n)$ для всіх $r \in \{0, 1, \ldots, s-1\}$

- Існує таке r, що $a^{(2r \times d)} \not\equiv \pm 1 \ (mod\ n)$, але $a^{(2(r+1) \times d)} \equiv 1 \ (mod\ n)$

Якщо жодна з цих умов не виконується, то n може бути простим числом (з певною ймовірністю).

Розглянемо алгоритм. Наведемо повний алгоритм тесту Міллера-Рабіна:

```
функція тест_міллера_рабіна(n, k):
    // n - число для перевірки
    // k - кількість раундів тестування

    // Крок 1: Обробка тривіальних випадків
    якщо n = 2:
        повернути "просте"
    якщо n < 2 або n парне:
        повернути "складене"

    // Крок 2: Представлення n-1 = 2^s × d
    s = 0
    d = n - 1
    поки d парне:
        d = d / 2
        s = s + 1

    // Крок 3: Виконання k раундів тестування
    для i від 1 до k:
        a = випадкове_число(2, n-2)
        якщо свідок_складеності(a, n, s, d):
            повернути "складене"

    повернути "ймовірно просте"
```

Функція перевірки свідка складеності

```
функція свідок_складеності(a, n, s, d):
    // Обчислення a^d mod n
    x = модульне_піднесення_до_степеня(a, d, n)

    якщо x = 1 або x = n - 1:
        повернути false

    // Піднесення до квадрату s-1 разів
    для r від 1 до s - 1:
        x = (x × x) mod n
        якщо x = n - 1:
            повернути false
```

```
    повернути true
```

Алгоритм модульного піднесення до степеня

Ефективне обчислення $a^b \ mod \ n$ є критичним для продуктивності тесту:

```
функція модульне_піднесення_до_степеня(a, b, n):
    результат = 1
    a = a mod n

    поки b > 0:
        якщо b непарне:
            результат = (результат × a) mod n
        b = b / 2
        a = (a × a) mod n

    повернути результат
```

Ймовірність помилки

Ключовою характеристикою тесту Міллера-Рабіна є його ймовірність помилки. Для складеного числа n ймовірність того, що випадково обраний свідок a не виявить його складеність, не перевищує 1/4. Отже, після k незалежних раундів тестування ймовірність помилково визначити складене число як просте не перевищує $(1/4)^k$.

Це означає, що вже при k = 10 ймовірність помилки становить менше 10^(-6), а при k = 20 - менше 10^(-12), що достатньо для більшості практичних застосувань.

Детерміністичні варіанти

Існують детерміністичні версії тесту Міллера-Рабіна для обмежених діапазонів чисел. Наприклад, для перевірки чисел до 2^{64} достатньо перевірити фіксований набір свідків: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37. Якщо число проходить тест для всіх цих свідків, воно гарантовано є простим.

Розглянемо детальний приклад перевірки числа n = 221 на простоту за допомогою тесту Міллера-Рабіна.

Крок 1: Представляємо n - 1 = 220 у вигляді

$2^s \times d: 220 = 4 \times 55 = 2^2 \times 55$

Отже, s = 2, d = 55.

Крок 2: Обираємо перший свідок a = 174.

Крок 3: Обчислюємо

$a^d \ mod \ n$: $174^{55} \ mod \ 221$

Використовуючи алгоритм швидкого піднесення до степеня:

- 55 = 110111_2 (двійкове представлення)

- Послідовно обчислюємо: $174^1, 174^2, 174^4, 174^8, 174^{16}, 174^{32}$ за модулем 221

- Комбінуємо результати: $174^{55} \equiv 47 \ (mod \ 221)$

Крок 4: Перевіряємо умови:

- $x_0 = 47 \neq 1$ і $x_0 \neq 220 \ (\neq -1 \ mod \ 221)$

- $x_1 = 47^2 \ mod \ 221 = 2209 \ mod \ 221 = 220 \equiv -1 \ (mod \ 221)$

Оскільки $x_1 \equiv -1 \ (mod \ 221)$, свідок 174 не доводить складеність числа 221.

Крок 5: Обираємо другий свідок а = 137.

Обчислюємо $137^{55} \ mod \ 221$:

- Результат: $137^{55} \equiv 188 \ (mod \ 221)$

Перевіряємо умови:

- $x_0 = 188 \neq 1$ і $x_0 \neq 220$

- $x_1 = 188^2 \ mod \ 221 = 35344 \ mod \ 221 = 205 \neq 1$ і $\neq 220$

- $x_2 = 205^2 \ mod \ 221 = 42025 \ mod \ 221 = 103 \neq 1$

Оскільки $x_1 \not\equiv \pm 1 \ (mod \ 221)$, але $x_2 \not\equiv 1 \ (mod \ 221)$, це не дає остаточного висновку. Однак, оскільки $x_2 = a^{(n-1)} \not\equiv 1 \ (mod \ 221)$, це порушує малу теорему Ферма, отже 221 є складеним числом.

Дійсно, $221 = 13 \times 17$.

Область застосування

Тест Міллера-Рабіна має широке застосування в різних галузях інформатики та криптографії:

Криптографія

- Генерація ключів RSA: При створенні RSA-ключів потрібно знайти два великих простих числа. Тест Міллера-Рабіна використовується для швидкої перевірки випадково згенерованих великих чисел на простоту.

- Криптографія на еліптичних кривих: Для забезпечення безпеки потрібно працювати з простими полями, що вимагає перевірки великих чисел на простоту.

- Протоколи обміну ключами: Багато протоколів, таких як Diffie-Hellman, вимагають використання великих простих чисел.

Теорія чисел та математика

- Пошук великих простих чисел: Тест використовується в проектах пошуку простих чисел спеціального вигляду, таких як числа Мерсенна.

- Факторизація чисел: Як допоміжний інструмент для відсіювання простих дільників.

- Дослідження розподілу простих чисел: Для статистичного аналізу простих чисел у великих діапазонах.

Комп'ютерні науки

- Хешування та структури даних: Деякі алгоритми хешування використовують прості числа для мінімізації колізій.

- Генерація псевдовипадкових чисел: Лінійні конгруентні генератори часто використовують великі прості модулі.

- Розподілені обчислення: Для розподілу навантаження часто використовуються хеш-функції з простими модулями.

Приклади застосування

Приклад 1: Генерація RSA-ключів

При генерації 2048-бітних RSA-ключів потрібно знайти два 1024-бітних простих числа. Процес виглядає так:

1. Генеруємо випадкове непарне 1024-бітне число n

2. Застосовуємо тест Міллера-Рабіна з k = 40 раундами

3. Якщо тест пройдено, використовуємо n; інакше переходимо до n + 2

4. Повторюємо процес для другого простого числа

З k = 40 ймовірність помилково прийняти складене число за просте становить менше $2^{(-80)}$, що набагато менше ймовірності апаратного збою.

Приклад 2: Пошук безпечних простих чисел

Безпечне просте число p - це таке просте число, що (p-1)/2 також є простим. Алгоритм пошуку:

```
функція знайти_безпечне_просте(біти):
    поки true:
        q = генерувати_випадкове_просте(біти - 1)
        p = 2 × q + 1
        якщо тест_міллера_рабіна(p, 20):
            повернути p
```

Приклад 3: Сертифікати простоти

Для критичних застосувань можна створити сертифікат простоти, комбінуючи тест Міллера-Рабіна з іншими методами:

1. Спочатку застосовуємо тест Міллера-Рабіна для швидкої перевірки

2. Якщо число ймовірно просте, застосовуємо детерміністичний тест (наприклад, AKS)

3. Зберігаємо результати проміжних обчислень як сертифікат

Переваги та недоліки

Переваги

- Висока швидкість: Часова складність $O(k \times \log^3 n)$, де k - кількість раундів. Це значно швидше за детерміністичні алгоритми.

- Простота реалізації: Алгоритм легко реалізувати, використовуючи лише базові арифметичні операції.

- Контрольована точність: Можна налаштувати кількість раундів для досягнення потрібного рівня надійності.

- Ефективність для великих чисел: Алгоритм добре масштабується для чисел з тисячами біт.

- Можливість розпаралелювання: Різні раунди можна виконувати незалежно на різних процесорах.

Недоліки

- Імовірнісна природа: Алгоритм не дає абсолютної гарантії простоти числа, лише високу ймовірність.

- Залежність від генератора випадкових чисел: Якість результатів залежить від якості випадкових чисел.

- Не дає факторизації: Якщо число складене, алгоритм не вказує його дільники.

- Вразливість до спеціальних чисел: Для чисел Кармайкла потрібно більше раундів тестування.

- Необхідність багатократного тестування: Для високої надійності потрібно виконати багато раундів.

Контрольні запитання

1. На яких математичних принципах базується тест Міллера-Рабіна?

2. Як представити n-1 у вигляді 2^s × d і навіщо це потрібно?

3. Що таке свідок складеності і як він визначається?

4. Яка ймовірність помилки тесту Міллера-Рабіна після k раундів?

5. Чим відрізняється детерміністична версія тесту від імовірнісної?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження тесту Міллера-Рабіна для перевірки чисел на простоту

2. Мета

Вивчити принципи роботи імовірнісних тестів на простоту, реалізувати тест Міллера-Рабіна, дослідити його ефективність та точність для різних класів чисел.

3. Завдання

- Реалізувати алгоритм швидкого модульного піднесення до степеня

- Реалізувати базовий тест Міллера-Рабіна

- Створити детерміністичну версію для малих чисел

- Реалізувати генератор простих чисел заданої довжини

- Провести статистичний аналіз ефективності алгоритму

- Порівняти з іншими тестами на простоту

4. Опис

Лабораторна робота передбачає повну реалізацію тесту Міллера-Рабіна з усіма допоміжними функціями. Необхідно створити модуль для роботи з великими числами, реалізувати ефективне модульне піднесення до степеня, основний алгоритм тесту з можливістю налаштування кількості раундів. Окремо слід реалізувати детерміністичну версію для чисел до 2^64. Програма повинна включати генератор простих чисел заданої бітової довжини, інструменти для тестування та аналізу продуктивності, візуалізацію процесу перевірки.

5. Порядок виконання

1. Реалізувати функцію модульного піднесення до степеня

2. Реалізувати розклад n-1 на 2^s × d

3. Реалізувати функцію перевірки свідка складеності

4. Об’єднати компоненти в повний тест Міллера-Рабіна

5. Додати детерміністичну версію з фіксованими свідками

6. Створити генератор простих чисел

7. Реалізувати інтерфейс користувача

8. Провести тестування та аналіз результатів

6. Результат

Програма повинна надавати можливість перевіряти довільні числа на простоту з вибором кількості раундів, автоматично генерувати прості числа заданої довжини, візуалізувати процес перевірки з покроковим виведенням проміжних результатів. Для аналізу ефективності програма повинна вимірювати час виконання для чисел різної довжини, оцінювати фактичну ймовірність помилки на тестових даних, порівнювати швидкість з іншими алгоритмами. Результати аналізу повинні бути представлені у вигляді графіків та таблиць.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функція модульного піднесення до степеня

- Функція розкладу n-1 на 2^s × d

- Функція перевірки свідка складеності

- Основна функція тесту Міллера-Рабіна

- Детерміністична версія алгоритму

- Генератор простих чисел

2. Продемонструвати роботу додатку:

- Перевірка конкретних чисел на простоту

- Генерація простих чисел різної довжини

- Візуалізація кроків алгоритму

- Аналіз часу виконання для різних розмірів чисел

- Статистика помилкових результатів

3. Бути готовим відповісти на питання:

- Чому тест називається імовірнісним?

- Як вибрати оптимальну кількість раундів?

- Що таке числа Кармайкла і як вони впливають на тест?

- Як працює детерміністична версія алгоритму?

- Які оптимізації можна застосувати для великих чисел?

Тести

1. Тест Міллера-Рабіна базується на:

1) Малій теоремі Ферма та властивостях квадратних коренів

2) Теоремі Ейлера

3) Китайській теоремі про залишки

4) Алгоритмі Евкліда

2. Для числа n представлення n-1 = 2^s × d означає, що:

1) d - непарне число

2) s - непарне число

3) d - парне число

4) s = d

3. Ймовірність помилки тесту Міллера-Рабіна після k раундів не перевищує:

1) 1/2^k

2) 1/4^k

3) 1/k

4) k/4

4. Свідок складеності для числа n - це таке число a, що:

1) Доводить простоту n

2) Доводить складеність n

3) НСД(a, n) > 1

4) a ділить n

5. Часова складність тесту Міллера-Рабіна з k раундами:

1) O(k × log n)

2) O(k × $\log^2$ n)

3) O(k × $\log^3$ n)

4) O(k × n)

Відповіді:

1-1,

2-1,

3-2,

4-2,

5-3

Тести підвищеної складності

1. Числа Кармайкла особливі тим, що:

1) Вони прості

2) Для них a^(n-1) ≡ 1 (mod n) для всіх a взаємно простих з n

3) Вони завжди парні

4) Тест Міллера-Рабіна не працює для них

2. Для детерміністичної версії тесту Міллера-Рабіна для чисел до 2^64 достатньо перевірити:

1) Один свідок

2) Всі числа від 2 до n-1

3) Фіксований набір з 12 свідків

4) Випадкові 64 свідки

3. Якщо при тестуванні a^d ≡ 1 (mod n), то:

1) n обов’язково просте

2) n обов’язково складене

3) a не є свідком складеності n

4) Потрібно перевірити інший свідок

4. Найгірший випадок для тесту Міллера-Рабіна:

1) Прості числа Мерсенна

2) Числа Кармайкла

3) Прості числа Ферма

4) Квадрати простих чисел

5. При генерації RSA-ключів рекомендована кількість раундів тесту:

1) 1-5

2) 10-20

3) 40-50

4) 100+

6. Сильне псевдопросте число за основою a - це складене n таке, що:

1) a^(n-1) ≡ 1 (mod n)

2) Проходить тест Міллера-Рабіна для свідка a

3) НСД(a, n) = 1

4) n ділить a^n - a

7. Для простого числа p властивість $x^2 \equiv 1 \pmod{p}$ означає:

1) $x \equiv 1 \pmod{p}$

2) $x \equiv -1 \pmod{p}$

3) $x \equiv \pm 1 \pmod{p}$

4) x може бути будь-яким

8. Оптимізація Монтгомері для модульного множення:

1) Зменшує кількість ділень

2) Збільшує точність

3) Працює тільки для простих модулів

4) Вимагає попереднього обчислення факторизації

9. Для числа n = 561 (найменше число Кармайкла) тест Міллера-Рабіна:

1) Завжди дає правильний результат

2) Ніколи не виявляє складеність

3) Виявляє складеність для більшості свідків

4) Працює тільки для парних свідків

10. Комбінування тесту Міллера-Рабіна з пробним діленням ефективне, коли:

1) Перевіряються дуже великі числа

2) Потрібна 100% гарантія

3) Числа мають малі прості дільники

4) Числа є степенями двійки

Відповіді:

1-2 Числа Кармайкла задовольняють умову Ферма для всіх взаємно простих з n чисел

2-3 Достатньо перевірити 12 конкретних свідків для гарантованого результату

3-3 Якщо перша перевірка дає 1, це не доводить складеність

4-2 Для чисел Кармайкла більше свідків не виявляють складеність

5-3 Для криптографічної стійкості рекомендується 40-50 раундів

6-2 Сильне псевдопросте проходить повний тест Міллера-Рабіна для даного свідка

7-3 Це основна властивість квадратних коренів з 1 за простим модулем

8-1 Алгоритм Монтгомері замінює ділення на зсуви та множення

9-3 Тест Міллера-Рабіна ефективний навіть для чисел Кармайкла

10-3 Пробне ділення швидко відсіює числа з малими дільниками

Завдання для самостійної роботи

1. Реалізуйте оптимізований тест Міллера-Рабіна з використанням алгоритму Монтгомері для модульного множення. Порівняйте швидкість з базовою реалізацією для чисел різної довжини.

2. Створіть програму для пошуку чисел Кармайкла в заданому діапазоні. Дослідіть, яка частка свідків не виявляє їх складеність у тесті Міллера-Рабіна.

3. Реалізуйте адаптивний алгоритм вибору кількості раундів залежно від розміру числа та необхідного рівня надійності.

4. Напишіть програму для генерації безпечних простих чисел (p таких, що (p-1)/2 також просте) з використанням тесту Міллера-Рабіна.

5. Створіть паралельну версію тесту Міллера-Рабіна, де різні раунди виконуються на різних процесорах або потоках.

6. Реалізуйте комбінований алгоритм, який спочатку застосовує пробне ділення на малі прості числа, потім тест Міллера-Рабіна.

7. Дослідіть розподіл свідків складеності для різних класів складених чисел. Створіть візуалізацію результатів.

8. Напишіть програму для створення та перевірки сертифікатів простоти, що поєднує тест Міллера-Рабіна з методом Прота або Люка.

9. Реалізуйте версію тесту для роботи з числами спеціального вигляду (Мерсенна, Ферма), використовуючи їх особливі властивості.

10. Створіть бенчмарк для порівняння різних тестів на простоту: Міллера-Рабіна, Соловея-Штрассена, Люка-Лемера для чисел Мерсенна.

Теми лабораторних робіт

1. Базова реалізація тесту Міллера-Рабіна та дослідження його точності

2. Оптимізація тесту Міллера-Рабіна для великих чисел з використанням алгоритму Монтгомері

3. Детерміністичні версії тесту Міллера-Рабіна для різних діапазонів чисел

4. Генератор криптографічно стійких простих чисел на основі тесту Міллера-Рабіна

5. Порівняльний аналіз імовірнісних тестів на простоту

6. Паралельна реалізація тесту Міллера-Рабіна для багатоядерних систем

7. Дослідження чисел Кармайкла та їх властивостей у контексті тестів на простоту

8. Комбіновані алгоритми перевірки на простоту з адаптивним вибором методу

9. Застосування тесту Міллера-Рабіна в криптографічних протоколах

10. Створення системи сертифікації простоти з використанням різних алгоритмів

Решето Ератосфена

Теоретичні відомості

Решето Ератосфена є одним з найдавніших і найефективніших алгоритмів для знаходження всіх простих чисел до заданого числа n. Цей алгоритм, названий на честь давньогрецького математика Ератосфена з Кирени, базується на простому принципі: якщо число p є простим, то всі його кратні 2p, 3p, 4p, ... є складеними числами.

Алгоритм працює методом послідовного відсіювання складених чисел. Спочатку створюється список всіх натуральних чисел від 2 до n. Потім, починаючи з найменшого числа в списку (яке є простим), викреслюються всі числа, кратні йому. Процес повторюється для наступного невикресленого числа, доки не будуть оброблені всі числа до $\sqrt{n}$. Числа, що залишилися невикресленими, є простими.

Математичне обґрунтування того, чому достатньо перевіряти числа лише до $\sqrt{n}$, полягає в наступному: якщо число m є складеним, то воно має принаймні один простий дільник $p \leqslant \sqrt{m}$. Дійсно, якщо $m = p \times q$ і обидва множники $p, q > \sqrt{m}$, то $p \times q > m$, що є протиріччям. Отже, при обробці всіх простих чисел до $\sqrt{n}$ ми гарантовано викреслимо всі складені числа до n.

Розглянемо покроковий алгоритм решета Ератосфена:

```
функція решето_ератосфена(n):
    // Створюємо масив для позначення простоти чисел
    є_простим[] = новий_булевий_масив[n + 1]

    // Ініціалізація: спочатку вважаємо всі числа простими
    для і від 0 до n:
        є_простим[i] = true

    // 0 та 1 не є простими числами
    є_простим[0] = false
    є_простим[1] = false

    // Основний цикл решета
    для p від 2 до √n:
        якщо є_простим[p]:
            // Викреслюємо всі кратні p
            для і від p × p до n з кроком p:
                є_простим[i] = false

    // Збираємо всі прості числа
    прості_числа = []
    для і від 2 до n:
        якщо є_простим[i]:
            додати і до прості_числа

    повернути прості_числа
```

Оптимізації алгоритму

Існує кілька важливих оптимізацій базового алгоритму решета Ератосфена:

Оптимізація 1: Початок викреслювання з p^2

Замість того, щоб починати викреслювання кратних числа p з $2p$, можна починати з p^2. Це пояснюється тим, що всі складені числа менші за p^2 вже були викреслені на попередніх ітераціях, оскільки вони мають менші прості дільники.

Оптимізація 2: Обробка лише непарних чисел

Оскільки всі парні числа, крім 2, є складеними, можна зменшити обсяг пам’яті вдвічі, зберігаючи інформацію лише про непарні числа:

```
функція решето_непарних(n):
    // Розмір масиву для непарних чисел
    розмір = (n - 1) / 2
    є_простим[] = новий_булевий_масив[розмір]

    // Ініціалізація
    для і від 0 до розмір - 1:
        є_простим[і] = true

    // Основний цикл для непарних чисел
    для і від 0 до розмір - 1:
        якщо є_простим[і]:
            // Обчислюємо значення непарного числа
            р = 2 × і + 3

            якщо р × р > n:
                перервати

            // Викреслюємо непарні кратні р
            для j від (р × р - 3) / 2 до розмір - 1 з кроком р:
                є_простим[j] = false

    // Збираємо прості числа
    прості_числа = [2]  // 2 - єдине парне просте число
    для і від 0 до розмір - 1:
        якщо є_простим[і]:
            додати (2 × і + 3) до прості_числа

    повернути прості_числа
```

Оптимізація 3: Сегментоване решето

Для дуже великих значень n може не вистачити пам’яті для зберігання всього масиву. Сегментоване решето обробляє числа блоками:

```
функція сегментоване_решето(n, розмір_сегмента):
    // Знаходимо всі прості до √n звичайним решетом
    межа = ⌊√n⌋
    базові_прості = решето_ератосфена(межа)

    всі_прості = []

    // Обробляємо сегменти
    для початок від 0 до n з кроком розмір_сегмента:
```

```
        кінець = мін(початок + розмір_сегмента - 1, n)

        // Масив для поточного сегмента
        є_простим[] = новий_булевий_масив[кінець - початок + 1]
        для і від 0 до кінець - початок:
            є_простим[і] = true

        // Викреслюємо кратні базових простих
        для кожного р в базові_прості:
            // Знаходимо перше кратне р в сегменті
            перше_кратне = максимум(р × р, (початок / р) × р)
            якщо перше_кратне < початок:
                перше_кратне = перше_кратне + р

            // Викреслюємо кратні
            для j від перше_кратне до кінець з кроком р:
                є_простим[j - початок] = false

        // Збираємо прості з сегмента
        для і від початок до кінець:
            якщо і > 1 та є_простим[і - початок]:
                додати і до всі_прості

    повернути всі_прості
```

Лінійне решето

Існує модифікація решета Ератосфена, яка працює за лінійний час O(n). Ця версія гарантує, що кожне складене число викреслюється рівно один раз:

```
функція лінійне_решето(n):
    є_простим[] = новий_булевий_масив[n + 1]
    прості = []
    найменший_простий_дільник[] = новий_масив[n + 1]

    для і від 2 до n:
        є_простим[і] = true

    для і від 2 до n:
        якщо є_простим[і]:
            прості.додати(і)
            найменший_простий_дільник[і] = і

        для j від 0 до прості.довжина - 1:
            якщо прості[j] > найменший_простий_дільник[і]:
                перервати
            якщо прості[j] × і > n:
```

```
            перервати

        є_простим[прості[j] × i] = false
        найменший_простий_дільник[прості[j] × i] = прості[j]

    повернути прості
```

Розглянемо детальний приклад роботи решета Ератосфена для знаходження всіх простих чисел до n = 30.

Крок 1: Ініціалізація Створюємо список чисел від 2 до 30: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30

Крок 2: p = 2 Перше невикреслене число - 2. Воно просте. Викреслюємо всі кратні 2, починаючи з $2^2 = 4$: 2, 3, 4, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19, ~~20~~, 21, ~~22~~, 23, ~~24~~, 25, ~~26~~, 27, ~~28~~, 29, ~~30~~

Крок 3: p = 3 Наступне невикреслене число - 3. Воно просте. Викреслюємо всі кратні 3, починаючи з $3^2 = 9$: 2, 3, 4, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~, ~~21~~, ~~22~~, 23, ~~24~~, 25, ~~26~~, ~~27~~, ~~28~~, 29, ~~30~~

Крок 4: p = 5 Наступне невикреслене число - 5. Воно просте. Викреслюємо всі кратні 5, починаючи з $5^2 = 25$: 2, 3, 4, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~, ~~21~~, ~~22~~, 23, ~~24~~, ~~25~~, ~~26~~, ~~27~~, ~~28~~, 29, ~~30~~

Крок 5: Завершення Оскільки $7^2 = 49 > 30$, алгоритм завершується. Невикреслені числа є простими: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29

Процес викреслювання можна представити у вигляді таблиці, де рядки відповідають ітераціям алгоритму:

```
Початок:  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30
p=2:      2  3  ×  5  ×  7  ×  9  × 11  × 13  × 15  × 17  × 19  × 21
× 23  × 25  × 27  × 29  ×
p=3:      2  3  ×  5  ×  7  ×  ×  × 11  × 13  × ×   × 17  × 19  × ×
× 23  × 25  × ×   × 29  ×
p=5:      2  3  ×  5  ×  7  ×  ×  × 11  × 13  × ×   × 17  × 19  × ×
× 23  × ×   × ×   × 29  ×
```

Область застосування

Решето Ератосфена має широке застосування в різних галузях математики та інформатики:

Криптографія

- Генерація простих чисел: Хоча для криптографічних застосувань потрібні дуже великі прості числа, решето Ератосфена використовується для

попередньої генерації малих простих чисел, які потім використовуються в тестах на простоту.

- Факторизація малих чисел: Решето можна модифікувати для збереження найменшого простого дільника кожного числа, що дозволяє швидко факторизувати числа в заданому діапазоні.

- Атаки на RSA: При малих значеннях приватного ключа решето може використовуватися для пошуку простих множників.

Теорія чисел

- Дослідження розподілу простих чисел: Решето дозволяє ефективно обчислювати функцію π(n) - кількість простих чисел, не більших за n.

- Пошук простих чисел-близнюків: Модифікація решета дозволяє знаходити пари простих чисел, що відрізняються на 2.

- Обчислення функції Ейлера: Решето можна адаптувати для обчислення φ(n) для всіх n в заданому діапазоні.

Алгоритмічні задачі

- Попередня обробка: У багатьох задачах потрібно знати всі прості числа до певної межі для подальших обчислень.

- Оптимізація алгоритмів: Знання простих чисел дозволяє оптимізувати алгоритми, що працюють з дільниками.

- Задачі на графах: Деякі задачі на графах використовують властивості простих чисел для оптимізації.

Приклади застосування

Приклад 1: Знаходження всіх простих дільників чисел від 1 до n

Модифікуємо решето для збереження найменшого простого дільника:

```
функція решето_з_дільниками(n):
    найменший_дільник[] = новий_масив[n + 1]

    для і від 0 до n:
        найменший_дільник[i] = i

    для р від 2 до √n:
        якщо найменший_дільник[р] = р:  // р - просте
            для і від р × р до n з кроком р:
                якщо найменший_дільник[i] = i:
                    найменший_дільник[i] = р
```

```
    повернути найменший_дільник
```

Приклад 2: Обчислення функції Ейлера для всіх чисел до n

```
функція обчислити_phi(n):
    phi[] = новий_масив[n + 1]

    для i від 0 до n:
        phi[i] = i

    для p від 2 до n:
        якщо phi[p] = p:  // p - просте
            для i від p до n з кроком p:
                phi[i] = phi[i] - phi[i] / p

    повернути phi
```

Приклад 3: Пошук простих чисел у заданому діапазоні [L, R]

Використовуємо сегментоване решето:

```
функція прості_в_діапазоні(L, R):
    // Знаходимо прості до √R
    межа = ⌊√R⌋
    базові_прості = решето_ератосфена(межа)

    // Масив для діапазону [L, R]
    є_простим[] = новий_булевий_масив[R - L + 1]
    для i від 0 до R - L:
        є_простим[i] = true

    // Спеціальний випадок для 1
    якщо L = 1:
        є_простим[0] = false

    // Викреслюємо кратні
    для кожного p в базові_прості:
        початок = максимум(p × p, ((L + p - 1) / p) × p)

        для j від початок до R з кроком p:
            є_простим[j - L] = false

    // Збираємо результат
    результат = []
    для i від L до R:
        якщо є_простим[i - L]:
            додати i до результат
```

```
    повернути результат
```

Переваги та недоліки

Переваги

- Простота реалізації: Алгоритм легко зрозуміти та реалізувати, потребує лише базових операцій з масивами.

- Ефективність для діапазону: При необхідності знайти всі прості числа в діапазоні, решето є найефективнішим методом.

- Передобчислення: Одноразове виконання решета дозволяє відповідати на багато запитів про простоту чисел за O(1).

- Можливість модифікації: Алгоритм легко адаптується для обчислення різних числових функцій.

- Оптимальна складність: Часова складність O(n log log n) близька до лінійної для практичних значень n.

Недоліки

- Використання пам’яті: Потребує O(n) пам’яті для зберігання масиву, що може бути проблемою для дуже великих n.

- Неефективність для окремих чисел: Якщо потрібно перевірити простоту лише одного великого числа, решето неефективне.

- Обмеження діапазону: Практично обмежене значеннями n до 10^8-10^9 через обмеження пам’яті.

- Складність оптимізацій: Ефективні версії (сегментоване, лінійне решето) значно складніші в реалізації.

- Послідовна природа: Важко ефективно розпаралелити базовий алгоритм.

Контрольні запитання

1. Чому в решеті Ератосфена достатньо перевіряти числа лише до $\sqrt{n}$?

2. Яка часова складність решета Ератосфена і від чого вона залежить?

3. Чому можна починати викреслювання кратних числа p з p^2, а не з 2p?

4. Як працює сегментоване решето і для чого воно використовується?

5. У чому основна відмінність лінійного решета від класичного?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму решета Ератосфена

2. Мета

Вивчити принципи роботи решета Ератосфена, реалізувати різні варіанти алгоритму, дослідити їх ефективність та застосувати для розв’язання практичних задач теорії чисел.

3. Завдання

- Реалізувати базовий алгоритм решета Ератосфена
- Реалізувати оптимізовану версію з обробкою лише непарних чисел
- Реалізувати сегментоване решето для великих діапазонів
- Модифікувати решето для обчислення функції Ейлера
- Створити візуалізацію процесу відсіювання
- Провести порівняльний аналіз різних реалізацій

4. Опис

Лабораторна робота передбачає поетапну реалізацію різних варіантів решета Ератосфена. Спочатку необхідно реалізувати базовий алгоритм з можливістю візуалізації процесу відсіювання. Далі слід реалізувати оптимізації: обробку лише непарних чисел та сегментоване решето. Окремим завданням є модифікація алгоритму для обчислення різних арифметичних функцій. Програма повинна включати інтерфейс для вибору типу решета, введення параметрів та відображення результатів у зручному вигляді.

5. Порядок виконання

1. Створити базову реалізацію решета Ератосфена
2. Додати візуалізацію процесу відсіювання
3. Реалізувати оптимізацію для непарних чисел
4. Реалізувати сегментоване решето
5. Створити модифікації для обчислення арифметичних функцій
6. Реалізувати інтерфейс користувача
7. Провести тестування та порівняльний аналіз
8. Оформити звіт з результатами

6. Результат

Програма повинна надавати можливість знаходити всі прості числа до заданого n різними методами, візуалізувати процес відсіювання покроково

або анімовано, обчислювати різні арифметичні функції на основі решета, працювати з великими діапазонами через сегментоване решето. Результати повинні включати список знайдених простих чисел, статистику виконання (час, використана пам’ять), порівняльні графіки для різних реалізацій. Програма повинна коректно обробляти граничні випадки та великі значення n.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функція базового решета Ератосфена
- Функція оптимізованого решета для непарних чисел
- Функція сегментованого решета
- Функція обчислення функції Ейлера через решето
- Модуль візуалізації процесу відсіювання
- Модуль порівняння продуктивності

2. Продемонструвати роботу додатку:

- Знаходження простих чисел до різних значень n
- Візуалізація процесу відсіювання
- Робота сегментованого решета для великих діапазонів
- Обчислення арифметичних функцій
- Графіки порівняння часу виконання та використання пам’яті

3. Бути готовим відповісти на питання:

- Чому складність решета O(n log log n)?
- Як вибрати оптимальний розмір сегмента?
- Які ще арифметичні функції можна обчислити через решето?
- Як розпаралелити решето Ератосфена?
- Які існують узагальнення решета для інших задач?

Тести

1. Часова складність решета Ератосфена:

1) O(n)

2) O(n log n)

3) O(n log log n)

4) $O(n^2)$

2. При викреслюванні кратних числа p можна починати з:

1) p

2) 2p

3) p^2

4) p^3

3. Для знаходження всіх простих до n достатньо перевірити числа до:

1) n/2

2) $\sqrt{n}$

3) n/log n

4) n

4. Основна перевага сегментованого решета:

1) Менша часова складність

2) Економія пам’яті

3) Простота реалізації

4) Краща точність

5. Лінійне решето гарантує, що кожне складене число:

1) Перевіряється один раз

2) Викреслюється один раз

3) Має один дільник

4) Є парним

Відповіді:

1-3,

2-3,

3-2,

4-2,

5-2

Тести підвищеної складності

1. При оптимізації решета для непарних чисел, індекс i відповідає числу:

1) i

2) $2i + 1$

3) $2i + 3$

4) $2i - 1$

2. Функція $\pi(n)$ (кількість простих $\leqslant n$) асимптотично дорівнює:

1) $n / \log n$

2) $n / \log^2 n$

3) $\sqrt{n}$

4) $n / 2$

3. Для обчислення функції Мебіуса через решето потрібно зберігати:

1) Всі прості дільники

2) Кількість простих дільників

3) Найбільший простий дільник

4) Степені простих дільників

4. Оптимальний розмір сегмента для сегментованого решета зазвичай:

1) $\sqrt{n}$

2) Розмір кешу L1

3) $n / \log n$

4) Константа

5. Решето Ератосфена можна модифікувати для знаходження:

1) Простих чисел виду $4k + 1$

2) Чисел Фібоначчі

3) Досконалих чисел

4) Факторіалів

6. При паралелізації решета найефективніше:

1) Розділити масив на частини

2) Розділити прості числа між потоками

3) Використати сегментоване решето

4) Паралелити викреслювання кратних

7. Для знаходження простих близнюків через решето потрібно:

1) Два окремих решета

2) Перевірка сусідніх елементів

3) Спеціальна модифікація алгоритму

4) Подвійний прохід

8. Квадратичне решето використовується для:

1) Знаходження квадратів простих

2) Факторизації великих чисел

3) Генерації простих чисел

4) Обчислення квадратних коренів

9. Решето Аткіна має складність:

1) O(n)

2) O(n / log log n)

3) O(n log log n)

4) $O(n^2)$

10. Для обчислення суми дільників через решето найкраще:

1) Зберігати всі дільники

2) Використати мультиплікативність функції

3) Застосувати формулу включень-виключень

4) Факторизувати кожне число

Відповіді:

1-3 Непарні числа 3, 5, 7, ... відповідають індексам 0, 1, 2, ..., тобто 2i+3

2-1 За теоремою про розподіл простих чисел $\pi(n) \sim n/\ln(n)$

3-2 Функція Мебіуса залежить від парності кількості простих дільників

4-2 Розмір кешу забезпечує оптимальну локальність даних

5-1 Можна модифікувати для пошуку простих спеціального вигляду

6-3 Сегменти можна обробляти незалежно різними потоками

7-2 Достатньо перевірити, чи числа i та i+2 обидва прості

8-2 Квадратичне решето - метод факторизації

9-2 Решето Аткіна має кращу асимптотику за рахунок складності

10-2 Використання мультиплікативності дозволяє ефективно обчислювати суму

Завдання для самостійної роботи

1. Реалізуйте решето Ератосфена з використанням бітового масиву для економії пам’яті. Порівняйте використання пам’яті з звичайною реалізацією.

2. Створіть паралельну версію сегментованого решета, де різні сегменти обробляються окремими потоками.

3. Модифікуйте решето для знаходження всіх простих чисел-близнюків (пар простих, що відрізняються на 2) до заданого n.

4. Реалізуйте решето для обчислення функції Мебіуса $\mu(n)$ для всіх чисел до n.

5. Напишіть програму, яка використовує решето для швидкої факторизації всіх чисел у заданому діапазоні.

6. Створіть візуалізацію спіралі Уляма, використовуючи решето Ератосфена для визначення простих чисел.

7. Реалізуйте решето Аткіна та порівняйте його продуктивність з решетом Ератосфена.

8. Модифікуйте решето для знаходження всіх напівпростих чисел (добутків двох простих) до n.

9. Створіть веб-додаток для інтерактивної візуалізації роботи різних варіантів решета.

10. Реалізуйте алгоритм для обчислення суми всіх простих чисел до 10^9, використовуючи сегментоване решето та формулу Мейселя-Лемера.

Теми лабораторних робіт

1. Базова реалізація решета Ератосфена та його візуалізація

2. Оптимізації решета Ератосфена: робота з пам’яттю та швидкістю

3. Сегментоване решето для роботи з великими діапазонами чисел

4. Модифікації решета для обчислення арифметичних функцій

5. Паралельні алгоритми на основі решета Ератосфена

6. Пошук простих чисел спеціального вигляду через решето

7. Порівняльний аналіз різних алгоритмів пошуку простих чисел

8. Застосування решета в задачах факторизації та криптографії

9. Візуалізація розподілу простих чисел за допомогою решета

10. Решето Ератосфена в задачах олімпіадного програмування

Решето Аткіна

Теоретичні відомості

Решето Аткіна є сучасним алгоритмом для знаходження всіх простих чисел до заданого числа n, розроблений математиками Артуром Аткіним та Даніелем Бернштейном у 2003 році. Цей алгоритм є теоретично швидшим за класичне решето Ератосфена, маючи часову складність O(n / log log n), хоча на практиці його перевага проявляється лише для дуже великих значень n.

Решето Аткіна базується на теорії квадратичних форм та властивостях простих чисел за різними модулями. Основна ідея полягає в тому, що всі прості числа, крім 2 та 3, належать до одного з класів лишків за модулем 60:

- 1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59 (mod 60)

Це випливає з того, що числа, кратні 2, 3 або 5, не можуть бути простими (крім самих 2, 3, 5), а $60 = 2^2 \times 3 \times 5$.

Алгоритм використовує три квадратичні форми для визначення потенційних простих чисел:

1. $\mathbf{4x^2 + y^2}$ - для чисел виду $n \equiv 1$ або 5 (mod 12)

2. $\mathbf{3x^2 + y^2}$ - для чисел виду $n \equiv 7$ (mod 12)

3. $\mathbf{3x^2 - y^2}$ - для чисел виду $n \equiv 11$ (mod 12), де $x > y$

Для кожної з трьох квадратичних форм існує критерій, що визначає простоту числа n:

Теорема 1: Непарне число n є простим тоді і тільки тоді, коли: - Для $n \equiv 1$ (mod 4): кількість розв'язків рівняння $4x^2 + y^2 = n$ є непарною - Для $n \equiv 3$ (mod 4): кількість розв'язків рівняння $3x^2 + y^2 = n$ є непарною

Теорема 2: Для чисел виду $n \equiv 11$ (mod 12), число n є простим тоді і тільки тоді, коли кількість розв'язків рівняння $3x^2 - y^2 = n$ (де $x > y$) є непарною.

Ці теореми випливають з теорії квадратичних форм та закону квадратичної взаємності.

Наведемо повний алгоритм решета Аткіна:

```
функція решето_аткіна(межа):
    // Ініціалізація
```

```
є_простим[] = новий_булевий_масив[межа + 1]
для і від 0 до межа:
    є_простим[і] = false

// Встановлюємо 2, 3, 5 як прості
якщо межа ≥ 2: є_простим[2] = true
якщо межа ≥ 3: є_простим[3] = true
якщо межа ≥ 5: є_простим[5] = true

// Обмеження для циклів
межа_х = ⌊√межа⌋

// Перша квадратична форма: 4х² + у²
для х від 1 до межа_х:
    для у від 1 до ⌊√(межа - 4х²)⌋:
        n = 4 × х² + у²
        якщо n ≤ межа:
            залишок = n mod 12
            якщо залишок = 1 або залишок = 5:
                є_простим[n] = не є_простим[n]

// Друга квадратична форма: 3х² + у²
для х від 1 до межа_х:
    для у від 1 до ⌊√(межа - 3х²)⌋:
        n = 3 × х² + у²
        якщо n ≤ межа:
            якщо n mod 12 = 7:
                є_простим[n] = не є_простим[n]

// Третя квадратична форма: 3х² - у²
для х від 1 до межа_х:
    для у від 1 до мін(х-1, ⌊√(3х² - межа)⌋):
        n = 3 × х² - у²
        якщо n ≤ межа:
            якщо n mod 12 = 11:
                є_простим[n] = не є_простим[n]

// Відсіювання квадратів простих чисел
для і від 5 до ⌊√межа⌋:
    якщо є_простим[і]:
        квадрат = і × і
        для j від квадрат до межа з кроком квадрат:
            є_простим[j] = false

повернути є_простим
```

Оптимізації алгоритму

Існує кілька важливих оптимізацій решета Аткіна:

Оптимізація 1: Робота з модулем 60

Замість перевірки всіх чисел, можна працювати лише з числами, що дають допустимі залишки за модулем 60:

```
функція решето_аткіна_мод60(межа):
    // Допустимі залишки за модулем 60
    допустимі = [1,7,11,13,17,19,23,29,31,37,41,43,47,49,53,59]

    // Масив для зберігання стану
    розмір = межа / 60 + 1
    є_простим[][] = новий_булевий_масив[розмір][16]

    // Відображення залишків на індекси
    індекс_залишку[] = новий_масив[60]
    для і від 0 до 15:
        індекс_залишку[допустимі[і]] = і
```

Оптимізація 2: Сегментоване решето Аткіна

Для великих значень n можна використовувати сегментований підхід:

```
функція сегментоване_решето_аткіна(початок, кінець):
    // Знаходимо прості до √кінець
    базові_прості = решето_аткіна(⌊√кінець⌋)

    // Обробляємо сегмент [початок, кінець]
    розмір = кінець - початок + 1
    є_простим[] = новий_булевий_масив[розмір]

    // Застосовуємо квадратичні форми для сегмента
    застосувати_квадратичні_форми_до_сегмента(початок, кінець, є_прос
тим)

    // Відсіюємо квадрати простих
    для р в базові_прості:
        якщо р ≥ 5:
            перше = максимум(р², ((початок + р² - 1) / р²) × р²)
            для і від перше до кінець з кроком р²:
                є_простим[і - початок] = false

    повернути є_простим
```

Оптимізація 3: Передобчислення меж

Для кожної квадратичної форми можна передобчислити точні межі:

```
функція обчислити_межі_для_форми1(n, x):
    // Для 4x² + y² ≤ n
    y_max = ⌊√(n - 4x²)⌋

    // Додаткові обмеження для модуля
    y_початок = 1
    якщо x mod 3 = 0:
        // Оптимізація для кратних 3
        y_початок = 1

    повернути (y_початок, y_max)
```

Розглянемо детальний приклад роботи решета Аткіна для знаходження всіх простих чисел до n = 50.

Крок 1: Ініціалізація Створюємо масив є_простим[51] та ініціалізуємо все як false. Встановлюємо є_простим[2] = true, є_простим[3] = true, є_простим[5] = true.

Крок 2: Перша квадратична форма $4x^2 + y^2$

Для x = 1: - y = 1: n = $4(1)^2 + 1^2 = 5$, 5 mod 12 = 5 ✓ → перемикаємо є_простим[5] - y = 2: n = $4(1)^2 + 2^2 = 8$, 8 mod 12 = 8 ✗ - y = 3: n = $4(1)^2 + 3^2 = 13$, 13 mod 12 = 1 ✓ → перемикаємо є_простим[13] - y = 4: n = $4(1)^2 + 4^2 = 20$, 20 mod 12 = 8 ✗ - y = 5: n = $4(1)^2 + 5^2 = 29$, 29 mod 12 = 5 ✓ → перемикаємо є_простим[29] - y = 6: n = $4(1)^2 + 6^2 = 40$, 40 mod 12 = 4 ✗ - y = 7: n = $4(1)^2 + 7^2 = 53 > 50$

Для x = 2: - y = 1: n = $4(2)^2 + 1^2 = 17$, 17 mod 12 = 5 ✓ → перемикаємо є_простим[17] - y = 2: n = $4(2)^2 + 2^2 = 20$, 20 mod 12 = 8 ✗ - y = 3: n = $4(2)^2 + 3^2 = 25$, 25 mod 12 = 1 ✓ → перемикаємо є_простим[25] - y = 4: n = $4(2)^2 + 4^2 = 32$, 32 mod 12 = 8 ✗ - y = 5: n = $4(2)^2 + 5^2 = 41$, 41 mod 12 = 5 ✓ → перемикаємо є_простим[41]

Для x = 3: - y = 1: n = $4(3)^2 + 1^2 = 37$, 37 mod 12 = 1 ✓ → перемикаємо є_простим[37] - y = 2: n = $4(3)^2 + 2^2 = 40$, 40 mod 12 = 4 ✗ - y = 3: n = $4(3)^2 + 3^2 = 45$, 45 mod 12 = 9 ✗

Крок 3: Друга квадратична форма $3x^2 + y^2$

Для x = 1: - y = 1: n = $3(1)^2 + 1^2 = 4$, 4 mod 12 = 4 ✗ - y = 2: n = $3(1)^2 + 2^2 = 7$, 7 mod 12 = 7 ✓ → перемикаємо є_простим[7] - y = 3: n = $3(1)^2 + 3^2 = 12$, 12 mod 12 = 0 ✗ - y = 4: n = $3(1)^2 + 4^2 = 19$, 19 mod 12 = 7 ✓ → перемикаємо є_простим[19] - y = 5: n = $3(1)^2 + 5^2 = 28$, 28 mod 12 = 4 ✗ - y = 6: n = $3(1)^2 + 6^2 = 39$, 39 mod 12 = 3 ✗ - y = 7: n = $3(1)^2 + 7^2 = 52 > 50$

Для x = 2: - y = 1: n = 3(2)² + 1² = 13, 13 mod 12 = 1 ✗ - y = 2: n = 3(2)² + 2² = 16, 16 mod 12 = 4 ✗ - y = 3: n = 3(2)² + 3² = 21, 21 mod 12 = 9 ✗ - y = 4: n = 3(2)² + 4² = 28, 28 mod 12 = 4 ✗ - y = 5: n = 3(2)² + 5² = 37, 37 mod 12 = 1 ✗ - y = 6: n = 3(2)² + 6² = 48, 48 mod 12 = 0 ✗

Для x = 3: - y = 1: n = 3(3)² + 1² = 28, 28 mod 12 = 4 ✗ - y = 2: n = 3(3)² + 2² = 31, 31 mod 12 = 7 ✓ → перемикаємо є_простим[31] - y = 3: n = 3(3)² + 3² = 36, 36 mod 12 = 0 ✗ - y = 4: n = 3(3)² + 4² = 43, 43 mod 12 = 7 ✓ → перемикаємо є_простим[43]

Крок 4: Третя квадратична форма 3x² - y²

Для x = 2, y = 1: - n = 3(2)² - 1² = 11, 11 mod 12 = 11 ✓ → перемикаємо є_простим[11]

Для x = 3: - y = 1: n = 3(3)² - 1² = 26, 26 mod 12 = 2 ✗ - y = 2: n = 3(3)² - 2² = 23, 23 mod 12 = 11 ✓ → перемикаємо є_простим[23]

Для x = 4: - y = 1: n = 3(4)² - 1² = 47, 47 mod 12 = 11 ✓ → перемикаємо є_простим[47] - y = 2: n = 3(4)² - 2² = 44, 44 mod 12 = 8 ✗ - y = 3: n = 3(4)² - 3² = 39, 39 mod 12 = 3 ✗

Крок 5: Відсіювання квадратів Перевіряємо квадрати простих чисел: - 5² = 25: є_простим[25] = false (було перемкнено на кроці 2) - 7² = 49: є_простим[49] = false (якщо було встановлено)

Результат: Прості числа до 50: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47

Область застосування

Решето Аткіна має специфічні області застосування, де його теоретичні переваги можуть бути реалізовані:

Обчислювальна теорія чисел

- Пошук великих простих чисел: Для дуже великих значень n (понад 10^9) решето Аткіна може бути ефективнішим за решето Ератосфена.

- Дослідження розподілу простих: Алгоритм дозволяє ефективно досліджувати властивості простих чисел у різних класах лишків.

- Криптографічні застосування: Генерація простих чисел для криптографічних протоколів, особливо коли потрібні прості спеціального вигляду.

Теоретичні дослідження

- Вивчення квадратичних форм: Алгоритм ілюструє зв’язок між простими числами та квадратичними формами.

- Аналітична теорія чисел: Використовується для перевірки гіпотез про розподіл простих чисел.

- Алгоритмічна складність: Служить прикладом алгоритму з субквадратичною складністю для задачі пошуку простих.

Спеціалізовані обчислення

- Паралельні обчислення: Структура алгоритму дозволяє ефективну паралелізацію.

- GPU-обчислення: Квадратичні форми добре підходять для обчислень на графічних процесорах.

- Розподілені системи: Можливість обробляти незалежні діапазони робить алгоритм придатним для розподілених обчислень.

Приклади застосування

Приклад 1: Пошук простих чисел спеціального вигляду

Модифікуємо решето Аткіна для пошуку простих чисел виду 4k + 1:

```
функція прості_виду_4k_плюс_1(межа):
    прості = решето_аткіна(межа)
    результат = []

    для і від 5 до межа з кроком 4:
        якщо прості[i]:
            додати і до результат

    повернути результат
```

Приклад 2: Обчислення суми квадратів

Використовуємо властивості квадратичних форм для представлення чисел як суми квадратів:

```
функція представлення_як_сума_квадратів(n):
    // Використовуємо першу квадратичну форму
    представлення = []

    для x від 1 до ⌊√(n/4)⌋:
        y² = n - 4x²
        якщо y² > 0 та є_повний_квадрат(y²):
            y = ⌊√y²⌋
            представлення.додати((2x, y))
```

```
    повернути представлення
```

Приклад 3: Паралельне решето Аткіна

```
функція паралельне_решето_аткіна(межа, кількість_потоків):
    // Розділяємо роботу між потоками
    результати[] = новий_масив[кількість_потоків]

    паралельно для і від 0 до кількість_потоків - 1:
        початок = і × межа / кількість_потоків
        кінець = (і + 1) × межа / кількість_потоків - 1
        результати[і] = обробити_діапазон(початок, кінець)

    // Об'єднуємо результати
    фінальний_результат = об'єднати_результати(результати)

    // Відсіюємо квадрати простих
    відсіяти_квадрати(фінальний_результат, межа)

    повернути фінальний_результат
```

Переваги та недоліки

Переваги

- Теоретично краща складність: O(n / log log n) порівняно з O(n log log n) для решета Ератосфена.

- Менше операцій для великих n: Для дуже великих значень n алгоритм виконує менше операцій.

- Природна паралелізація: Три квадратичні форми можна обчислювати незалежно.

- Цікаві математичні властивості: Алгоритм демонструє глибокі зв’язки між простими числами та квадратичними формами.

- Можливість оптимізації для спеціальних випадків: Легко модифікується для пошуку простих чисел певного вигляду.

Недоліки

- Складність реалізації: Алгоритм значно складніший за решето Ератосфена.

- Більша константа: На практиці для помірних значень n решето Ератосфена часто швидше через меншу константу в O-нотації.

- Складніші оптимізації: Ефективні реалізації вимагають глибокого розуміння математичних властивостей.

- Більше операцій над пам’яттю: Перемикання бітів може бути менш ефективним за просте встановлення значень.

- Необхідність фінального відсіювання: Додатковий крок відсіювання квадратів ускладнює алгоритм.

Контрольні запитання

1. Які три квадратичні форми використовує решето Аткіна і для яких класів чисел?

2. Чому решето Аткіна працює лише з певними класами лишків за модулем 60?

3. Яка теоретична часова складність решета Аткіна і чим вона відрізняється від решета Ератосфена?

4. Що означає “перемикання” стану числа в решеті Аткіна?

5. Чому необхідний крок відсіювання квадратів простих чисел?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження решета Аткіна для пошуку простих чисел

2. Мета

Вивчити принципи роботи решета Аткіна, реалізувати алгоритм з оптимізаціями, порівняти його ефективність з решетом Ератосфена та дослідити можливості паралелізації.

3. Завдання

- Реалізувати базовий алгоритм решета Аткіна

- Реалізувати оптимізовану версію з роботою за модулем 60

- Створити паралельну версію алгоритму

- Реалізувати сегментоване решето Аткіна

- Провести порівняльний аналіз з решетом Ератосфена

- Візуалізувати процес роботи алгоритму

4. Опис

Лабораторна робота передбачає глибоке вивчення решета Аткіна через поетапну реалізацію. Спочатку необхідно реалізувати базову версію з трьома квадратичними формами. Далі слід оптимізувати алгоритм для роботи з модулем 60. Окремим завданням є реалізація паралельної версії,

що використовує можливості багатоядерних процесорів. Програма повинна включати детальну візуалізацію процесу відмічання чисел для кожної квадратичної форми та порівняння продуктивності з решетом Ератосфена.

5. Порядок виконання

1. Реалізувати три функції для квадратичних форм

2. Створити основний цикл решета Аткіна

3. Додати крок відсіювання квадратів

4. Реалізувати оптимізацію з модулем 60

5. Створити паралельну версію

6. Реалізувати сегментований варіант

7. Створити візуалізацію та інтерфейс

8. Провести тестування та аналіз продуктивності

6. Результат

Програма повинна надавати можливість знаходити всі прості числа до заданого n різними варіантами решета Аткіна, візуалізувати процес застосування кожної квадратичної форми, показувати статистику перемикань для кожного числа, порівнювати час виконання з решетом Ератосфена. Результати повинні включати графіки залежності часу виконання від n, аналіз ефективності паралелізації, демонстрацію роботи для різних діапазонів чисел.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функції для трьох квадратичних форм

- Основний алгоритм решета Аткіна

- Функція відсіювання квадратів

- Оптимізована версія з модулем 60

- Паралельна реалізація

- Модуль візуалізації

2. Продемонструвати роботу додатку:

- Знаходження простих чисел різними версіями алгоритму

- Візуалізація застосування квадратичних форм

- Порівняння швидкості з решетом Ератосфена
- Ефективність паралельної версії
- Робота сегментованого решета для великих діапазонів

3. Бути готовим відповісти на питання:
- Чому використовуються саме ці три квадратичні форми?
- Як пов’язані класи лишків з властивостями простих чисел?
- При яких значеннях n решето Аткіна стає ефективнішим?
- Як оптимізувати використання пам’яті в алгоритмі?
- Які можливості для подальших оптимізацій?

Тести

1. Решето Аткіна має теоретичну складність:

1) O(n)

2) O(n log log n)

3) O(n / log log n)

4) O(√n)

2. Скільки квадратичних форм використовує решето Аткіна?

1) 1

2) 2

3) 3

4) 4

3. Решето Аткіна працює з числами, що дають певні залишки за модулем:

1) 30

2) 60

3) 12

4) 24

4. Для числа $n \equiv 1 \pmod{12}$ використовується форма:

1) $4x^2 + y^2$

2) $3x^2 + y^2$

3) $3x^2 - y^2$

4) $x^2 + y^2$

5. Фінальний крок решета Аткіна:

1) Перевірка парності

2) Відсіювання квадратів простих

3) Сортування результату

4) Перевірка модулів

Відповіді:

1-3,

2-3,

3-2,

4-1,

5-2

Тести підвищеної складності

1. Для форми $3x^2 - y^2$ необхідна додаткова умова:

1) $x > y$

2) $x < y$

3) $x = y$

4) $x \geqslant 2y$

2. Кількість допустимих залишків за модулем 60:

1) 8

2) 12

3) 16

4) 20

3. Решето Аткіна “перемикає” стан числа, що означає:

1) Встановлює як просте

2) Встановлює як складене

3) Змінює стан на протилежний

4) Видаляє з розгляду

4. Теоретична перевага $O(n / \log \log n)$ проявляється при n більше:

1) 10^3

2) 10^6

3) 10^9

4) 10^{12}

5. Для паралелізації решета Аткіна найкраще:

1) Розділити масив на частини

2) Розділити квадратичні форми

3) Розділити за модулем

4) Використати конвеєр

6. Зв’язок решета Аткіна з законом квадратичної взаємності проявляється через:

1) Вибір модуля 60

2) Критерії простоти для квадратичних форм

3) Відсіювання квадратів

4) Початкові значення

7. Для оптимізації пам’яті в решеті Аткіна використовують:

1) Стиснення даних

2) Роботу лише з допустимими залишками

3) Динамічні структури

4) Кешування

8. Основна складність ефективної реалізації решета Аткіна:

1) Обчислення квадратів

2) Правильні межі циклів

3) Перевірка модулів

4) Виділення пам’яті

9. Решето Аткіна краще підходить для:

1) Малих діапазонів

2) Пошуку окремих простих

3) Великих суцільних діапазонів

4) Простих спеціального вигляду

10. Модифікація решета Аткіна для пошуку простих-близнюків:

1) Неможлива

2) Вимагає додаткової квадратичної форми

3) Потребує постобробки результатів

4) Працює автоматично

Відповіді:

1-1 Умова x > y необхідна для форми $3x^2 - y^2$

2-3 Існує 16 допустимих залишків за модулем 60

3-3 Алгоритм використовує XOR для зміни стану

4-3 Практична перевага проявляється для дуже великих n

5-2 Три форми можна обчислювати незалежно

6-2 Критерії базуються на квадратичній взаємності

7-2 Зберігаємо лише числа з допустимими залишками

8-2 Правильний розрахунок меж критичний для коректності

9-3 Алгоритм оптимізований для великих діапазонів

10-3 Потрібно перевірити сусідні прості числа

Завдання для самостійної роботи

1. Реалізуйте решето Аткіна з використанням SIMD-інструкцій процесора для прискорення обчислень квадратичних форм.

2. Створіть гібридний алгоритм, що використовує решето Аткіна для великих діапазонів та решето Ератосфена для малих.

3. Модифікуйте решето Аткіна для пошуку простих чисел Софі Жермен (p таких, що 2p + 1 також просте).

4. Реалізуйте розподілену версію решета Аткіна для обчислювального кластера з використанням MPI.

5. Дослідіть можливість використання інших квадратичних форм для покращення алгоритму.

6. Створіть GPU-реалізацію решета Аткіна з використанням CUDA або OpenCL.

7. Розробіть адаптивний алгоритм, що автоматично вибирає між решетом Аткіна та Ератосфена залежно від параметрів.

8. Реалізуйте решето Аткіна для знаходження простих у арифметичних прогресіях.

9. Створіть візуалізацію, що показує розподіл простих чисел за класами лишків модуля 60.

10. Дослідіть можливість комбінування решета Аткіна з тестами на простоту для дуже великих чисел.

Теми лабораторних робіт

1. Базова реалізація решета Аткіна та аналіз його складності
2. Оптимізація решета Аткіна для роботи з модулем 60
3. Паралельна реалізація решета Аткіна на багатоядерних системах
4. Порівняльний аналіз решета Аткіна та решета Ератосфена
5. GPU-прискорення решета Аткіна
6. Сегментоване решето Аткіна для надвеликих діапазонів
7. Гібридні алгоритми на основі решета Аткіна
8. Застосування решета Аткіна в криптографічних задачах
9. Дослідження квадратичних форм у контексті решета Аткіна
10. Розподілені обчислення з використанням решета Аткіна

12.2. Криптографія

Криптографія є наріжним каменем сучасної інформаційної безпеки, забезпечуючи конфіденційність, цілісність та автентичність даних у цифровому світі. У цьому розділі ми розглянемо фундаментальні криптографічні алгоритми та протоколи, які формують основу захисту інформації в сучасних комп'ютерних системах.

Розпочнемо з асиметричної криптографії, де різні ключі використовуються для шифрування та дешифрування. Алгоритм RSA, названий на честь Рівеста, Шаміра та Адлемана, став першим практичним методом шифрування з відкритим ключем і досі залишається одним з найпоширеніших. Ми детально вивчимо математичні основи RSA, включаючи використання великих простих чисел та модульної арифметики, а також розглянемо сучасні атаки та методи захисту.

Протокол Діффі-Геллмана революціонізував криптографію, вирішивши проблему безпечного обміну ключами через незахищені канали зв'язку. Ми дослідимо елегантну математичну конструкцію цього протоколу, яка

базується на складності задачі дискретного логарифмування, та його численні варіації і застосування в сучасних протоколах безпеки.

Криптографія на еліптичних кривих представляє наступне покоління асиметричних криптосистем, пропонуючи той самий рівень безпеки при значно менших розмірах ключів. Ми вивчимо математику еліптичних кривих, алгоритми ECDH для обміну ключами та ECDSA для цифрових підписів, а також розглянемо їх переваги в контексті обмежених обчислювальних ресурсів та мобільних пристроїв.

Криптографічні хеш-функції є критично важливими компонентами багатьох криптографічних протоколів. Ми детально розглянемо сімейство SHA-2, включаючи SHA-256 та SHA-512, новітній стандарт SHA-3, що базується на принципово іншій конструкції губки, та високопродуктивну функцію BLAKE2. Для кожної з них ми вивчимо внутрішню структуру, властивості безпеки та практичні застосування.

Завершимо розділ вивченням симетричного шифрування на прикладі AES-GCM - комбінації стандарту шифрування AES з режимом автентифікованого шифрування Galois/Counter Mode. Ця криптосистема забезпечує як конфіденційність, так і цілісність даних, що робить її ідеальним вибором для захисту даних у сучасних мережевих протоколах.

Кожен алгоритм буде розглянуто не лише з теоретичної точки зору, але й з практичної: ми реалізуємо ключові компоненти, проаналізуємо їх продуктивність, дослідимо вразливості та методи захисту. Особливу увагу приділимо правильному використанню криптографічних примітивів, оскільки навіть найсильніші алгоритми можуть бути скомпрометовані через помилки в реалізації або неправильне застосування.

Цей розділ закладе міцний фундамент для розуміння сучасної криптографії та підготує читача до роботи з криптографічними системами в реальних проектах, від захисту веб-застосунків до розробки безпечних протоколів зв’язку.

Криптосистема RSA

Теоретичні відомості

RSA є однією з найважливіших криптосистем з відкритим ключем, розробленою Роном Рівестом, Аді Шаміром та Леонардом Адлеманом у 1977

році. Ця криптосистема базується на складності факторизації великих складених чисел та забезпечує як шифрування, так і цифрові підписи.

Математичні основи RSA

Безпека RSA ґрунтується на кількох фундаментальних математичних концепціях. Основною є складність розкладання великого числа на прості множники. Якщо маємо число n = p × q, де p та q - великі прості числа, то знаючи n, обчислювально складно знайти p та q.

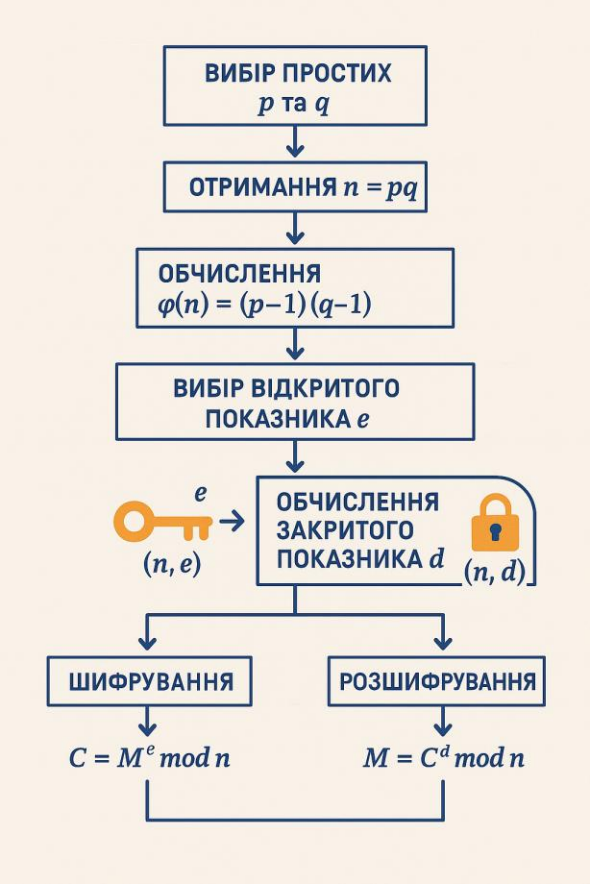
Другою важливою концепцією є функція Ейлера φ(n), яка для добутку двох простих чисел обчислюється як φ(n) = (p-1)(q-1). Ця функція визначає кількість чисел, менших за n та взаємно простих з n.

Третьою основою є мала теорема Ферма та її узагальнення - теорема Ейлера.

Якщо НСД(a, n) = 1, то $a^{\varphi(n)} \equiv 1 \ (mod\ n)$.

Це означає, що при піднесенні числа a до степеня $\varphi(n)$ за модулем n отримуємо 1.

Схема роботи алгоритма така:



Генерація ключів RSA

Процес генерації ключів RSA складається з таких кроків:

```
функція генерація_ключів_RSA(розмір_ключа):
    // Крок 1: Генерація двох великих простих чисел
    p = генерувати_просте_число(розмір_ключа / 2)
    q = генерувати_просте_число(розмір_ключа / 2)

    // Перевірка, що p ≠ q
    поки p = q:
        q = генерувати_просте_число(розмір_ключа / 2)

    // Крок 2: Обчислення модуля
    n = p × q

    // Крок 3: Обчислення функції Ейлера
    φ_n = (p - 1) × (q - 1)

    // Крок 4: Вибір відкритої експоненти e
    e = 65537  // Стандартне значення

    // Перевірка НСД(e, φ_n) = 1
    якщо НСД(e, φ_n) ≠ 1:
        e = знайти_взаємно_просте(φ_n)

    // Крок 5: Обчислення закритої експоненти d
    d = обернений_елемент(e, φ_n)

    // Формування ключів
    відкритий_ключ = (n, e)
    закритий_ключ = (n, d)

    // Додатково можна зберігати p, q для оптимізації
    закритий_ключ_розширений = (n, d, p, q)

    повернути (відкритий_ключ, закритий_ключ)
```

Генерація простих чисел для RSA

Для RSA потрібні великі прості числа. Алгоритм їх генерації такий:

```
функція генерувати_просте_число(біти):
    поки істина:
        // Генеруємо випадкове непарне число
        кандидат = випадкове_число(2^(біти-1), 2^біти - 1)
        якщо кандидат парне:
            кандидат = кандидат + 1
```

```
        // Швидка перевірка подільності на малі прості
        якщо швидка_перевірка(кандидат):
            // Тест Міллера-Рабіна
            якщо тест_міллера_рабіна(кандидат, 40):
                повернути кандидат

        кандидат = кандидат + 2
```

Обчислення оберненого елемента

Для знаходження закритої експоненти d використовується розширений алгоритм Евкліда:

```
функція обернений_елемент(a, m):
    // Розширений алгоритм Евкліда
    старий_r = a
    r = m
    старий_s = 1
    s = 0

    поки r ≠ 0:
        частка = старий_r div r

        // Оновлення r
        тимчасове = r
        r = старий_r - частка × r
        старий_r = тимчасове

        // Оновлення s
        тимчасове = s
        s = старий_s - частка × s
        старий_s = тимчасове

    // Перевірка існування оберненого
    якщо старий_r > 1:
        повернути помилка("Обернений елемент не існує")

    // Нормалізація результату
    якщо старий_s < 0:
        старий_s = старий_s + m

    повернути старий_s
```

Шифрування та дешифрування

Основні операції RSA - це модульне піднесення до степеня:

```
функція шифрування_RSA(повідомлення, відкритий_ключ):
    (n, e) = відкритий_ключ
```

```
    // Перевірка, що повідомлення < n
    якщо повідомлення ≥ n:
        повернути помилка("Повідомлення занадто велике")

    // Шифрування: c = m^e mod n
    шифротекст = модульне_піднесення(повідомлення, e, n)

    повернути шифротекст

функція дешифрування_RSA(шифротекст, закритий_ключ):
    (n, d) = закритий_ключ

    // Дешифрування: m = c^d mod n
    повідомлення = модульне_піднесення(шифротекст, d, n)

    повернути повідомлення
```

Швидке модульне піднесення до степеня

Ефективне обчислення $a^b \ mod \ n$ критично важливе для RSA:

```
функція модульне_піднесення(основа, експонента, модуль):
    результат = 1
    основа = основа mod модуль

    поки експонента > 0:
        // Якщо експонента непарна
        якщо експонента mod 2 = 1:
            результат = (результат × основа) mod модуль

        // Подвоюємо основу та половинимо експоненту
        експонента = експонента div 2
        основа = (основа × основа) mod модуль

    повернути результат
```

Оптимізація з використанням китайської теореми про залишки

Для прискорення дешифрування використовується китайська теорема про залишки:

```
функція дешифрування_RSA_CRT(шифротекст, розширений_ключ):
    (n, d, p, q) = розширений_ключ

    // Передобчислення
    dp = d mod (p - 1)
    dq = d mod (q - 1)
    qinv = обернений_елемент(q, p)
```

```
    // Обчислення за модулями p та q
    m1 = модульне_піднесення(шифротекст mod p, dp, p)
    m2 = модульне_піднесення(шифротекст mod q, dq, q)

    // Відновлення за китайською теоремою
    h = (qinv × (m1 - m2)) mod p
    якщо h < 0:
        h = h + p

    повідомлення = m2 + h × q

    повернути повідомлення
```

Схеми доповнення

Для безпечного використання RSA необхідно використовувати схеми доповнення:

```
функція PKCS1_v1_5_pad(повідомлення, розмір_ключа):
    // Розмір доповнення
    розмір_повідомлення = довжина(повідомлення)
    розмір_доповнення = розмір_ключа - розмір_повідомлення - 3

    якщо розмір_доповнення < 8:
        повернути помилка("Повідомлення занадто довге")

    // Генерація випадкового доповнення (без нулів)
    доповнення = []
    для і від 0 до розмір_доповнення - 1:
        байт = випадкове_число(1, 255)
        доповнення.додати(байт)

    // Формування блоку: 0x00 || 0x02 || доповнення || 0x00 || повідо
млення
    блок = [0x00, 0x02] + доповнення + [0x00] + повідомлення

    повернути блок
```

OAEP (Optimal Asymmetric Encryption Padding)

Сучасніша схема доповнення OAEP:

```
функція OAEP_encode(повідомлення, параметри):
    // Параметри
    k = параметри.розмір_ключа_байти
    mLen = довжина(повідомлення)
    hLen = параметри.розмір_хешу
```

```
// Перевірка довжини
якщо mLen > k - 2 × hLen - 2:
    повернути помилка("Повідомлення занадто довге")

// Крок 1: Доповнення повідомлення
PS = масив_нулів(k - mLen - 2 × hLen - 2)
DB = параметри.lHash + PS + [0x01] + повідомлення

// Крок 2: Генерація випадкової маски
seed = випадкові_байти(hLen)

// Крок 3: Маскування
dbMask = MGF(seed, k - hLen - 1)
maskedDB = XOR(DB, dbMask)

seedMask = MGF(maskedDB, hLen)
maskedSeed = XOR(seed, seedMask)

// Крок 4: Формування результату
EM = [0x00] + maskedSeed + maskedDB

повернути EM
```

Розглянемо детальний приклад з малими числами для демонстрації роботи RSA.

Крок 1: Генерація ключів

Виберемо малі прості числа для прикладу: -

p = 61 - q = 53

Обчислюємо модуль:

- $n = p \times q = 61 \times 53 = 3233$

Обчислюємо функцію Ейлера:

- $\varphi(n) = (p-1) \times (q-1) = 60 \times 52 = 3120$

Вибираємо відкриту експоненту:

- e = 17 (взаємно проста з 3120)

Знаходимо закриту експоненту:

- $d \equiv e^{-1} \pmod{\varphi(n)}$
    - $17 \times d \equiv 1 \pmod{3120}$

Використовуючи розширений алгоритм Евкліда: d = 2753

Перевірка: $17 \times 2753 = 46801 = 15 \times 3120 + 1 \equiv 1 \pmod{3120}$ ✓

Крок 2: Шифрування

Зашифруємо повідомлення M = 123:

- C = M^e mod n = 123^17 mod 3233

Обчислюємо за допомогою швидкого піднесення до степеня:

- 17 = 10001_2 (двійкове представлення)
- $123^1 \bmod 3233 = 123$
- $123^2 \bmod 3233 = 15129 \bmod 3233 = 2330$
- $123^4 \bmod 3233 = 2330^2 \bmod 3233 = 5428900 \bmod 3233 = 487$
- $123^8 \bmod 3233 = 487^2 \bmod 3233 = 237169 \bmod 3233 = 1095$
- $123^{16} \bmod 3233 = 1095^2 \bmod 3233 = 1199025 \bmod 3233 = 2900$

$C = 123^{16} \times 123^1 \bmod 3233 = 2900 \times 123 \bmod 3233 = 356700 \bmod 3233 = 855$

Крок 3: Дешифрування

Розшифруємо C = 855:

- M = C^d mod n = 855^2753 mod 3233

Використовуючи китайську теорему про залишки для оптимізації:

- dp = d mod (p-1) = 2753 mod 60 = 53
- dq = d mod (q-1) = 2753 mod 52 = 1
- qinv = $q^{-1} \bmod p = 53^{-1} \bmod 61 = 38$

Обчислюємо:

- m1 = 855^53 mod 61 = 1
- m2 = 855^1 mod 53 = 855 mod 53 = 12

Відновлюємо:

- h = (qinv × (m1 - m2)) mod p = (38 × (1 - 12)) mod 61 = (38 × (-11)) mod 61 = 50 - M = m2 + h × q = 12 + 50 × 53 = 12 + 2650 = 2662 mod 3233 = 123 ✓

Область застосування

RSA має широке застосування в різних сферах інформаційної безпеки:

Захист даних

- SSL/TLS протоколи: RSA використовується для початкового обміну ключами та автентифікації серверів у HTTPS з’єднаннях.

- Електронна пошта: Протоколи S/MIME та PGP використовують RSA для шифрування та підписування електронних листів.

- VPN з’єднання: Багато VPN протоколів використовують RSA для автентифікації та обміну сесійними ключами.

Цифрові підписи

- Документообіг: RSA забезпечує юридично значущі електронні підписи для документів.

- Оновлення програмного забезпечення: Виробники ПЗ використовують RSA для підписування оновлень.

- Сертифікати: Інфраструктура відкритих ключів PKI базується на RSA сертифікатах.

Автентифікація

- Двофакторна автентифікація: RSA токени генерують одноразові паролі.

- Смарт-карти: Банківські та ідентифікаційні картки використовують RSA для безпечної автентифікації.

- SSH: Протокол SSH підтримує RSA ключі для безпарольного входу.

Приклади застосування

Приклад 1: Гібридне шифрування

У реальних системах RSA комбінується з симетричним шифруванням:

```
функція гібридне_шифрування(дані, відкритий_ключ_RSA):
    // Генерація випадкового симетричного ключа
    симетричний_ключ = генерувати_AES_ключ(256)

    // Шифрування даних симетричним алгоритмом
    зашифровані_дані = AES_шифрування(дані, симетричний_ключ)

    // Шифрування симетричного ключа за допомогою RSA
    зашифрований_ключ = RSA_шифрування(симетричний_ключ, відкритий_кл
юч_RSA)

    // Повертаємо обидві частини
    повернути (зашифрований_ключ, зашифровані_дані)
```

Приклад 2: Цифровий підпис з хешуванням

```
функція створити_підпис_RSA(документ, закритий_ключ):
    // Обчислення хешу документа
    хеш = SHA256(документ)
```

```
    // Додавання структури PKCS#1
    структурований_хеш = додати_DigestInfo(хеш, "SHA256")

    // Доповнення за PKCS#1
    доповнений = PKCS1_pad_для_підпису(структурований_хеш)

    // Створення підпису
    підпис = RSA_операція_закритим_ключем(доповнений, закритий_ключ)

    повернути підпис
```

Приклад 3: Сліпий підпис RSA

Сліпі підписи використовуються для анонімних транзакцій:

```
функція сліпий_підпис_RSA(повідомлення, відкритий_ключ, закритий_ключ
):
    (n, e) = відкритий_ключ
    (n, d) = закритий_ключ

    // Клієнт: засліплення повідомлення
    r = випадкове_взаємно_просте(n)
    засліплене = (повідомлення × r^e) mod n

    // Сервер: підписування засліпленого повідомлення
    сліпий_підпис = модульне_піднесення(засліплене, d, n)

    // Клієнт: зняття засліплення
    r_inv = обернений_елемент(r, n)
    підпис = (сліпий_підпис × r_inv) mod n

    повернути підпис
```

Переваги та недоліки

Переваги

- Асиметрична природа: Не потребує безпечного каналу для обміну ключами, на відміну від симетричних алгоритмів.

- Подвійне призначення: Один алгоритм забезпечує як шифрування, так і цифрові підписи.

- Математична простота: Базується на зрозумілих математичних принципах, що полегшує аналіз безпеки.

- Довга історія: Понад 40 років успішного використання та ретельного криптоаналізу.

- Стандартизація: Широка підтримка в стандартах та протоколах безпеки.

Недоліки

- Швидкість: RSA значно повільніший за симетричні алгоритми через складні математичні операції.

- Розмір ключів: Для сучасного рівня безпеки потрібні ключі 2048-4096 біт, що набагато більше за симетричні.

- Вразливість до квантових комп'ютерів: Алгоритм Шора може ефективно факторизувати великі числа на квантовому комп'ютері.

- Складність безпечної реалізації: Потребує правильного доповнення та захисту від атак по сторонніх каналах.

- Обмеження на розмір повідомлення: Може шифрувати лише повідомлення, менші за модуль n.

Контрольні запитання

1. На якій математичній проблемі базується безпека RSA?

2. Чому для RSA важливо, щоб p та q були різними простими числами?

3. Як китайська теорема про залишки прискорює дешифрування в RSA?

4. Чому необхідно використовувати схеми доповнення при шифруванні RSA?

5. Яка роль функції Ейлера в алгоритмі RSA?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження криптосистеми RSA

2. Мета

Вивчити принципи роботи криптосистеми RSA, реалізувати основні компоненти алгоритму, дослідити вплив розміру ключів на безпеку та продуктивність.

3. Завдання

- Реалізувати генерацію простих чисел для RSA

- Реалізувати генерацію пари ключів RSA

- Створити функції шифрування та дешифрування

- Реалізувати оптимізацію з використанням CRT

- Реалізувати схему доповнення PKCS#1 v1.5

- Створити систему цифрового підпису

- Провести аналіз продуктивності та безпеки

4. Опис

Лабораторна робота передбачає повну реалізацію криптосистеми RSA від генерації ключів до практичного застосування. Необхідно створити модуль для роботи з великими числами, реалізувати алгоритми перевірки простоти та генерації простих чисел. Окремо слід реалізувати оптимізації для підвищення швидкодії. Програма повинна демонструвати як базове шифрування, так і практичні застосування - гібридне шифрування та цифрові підписи.

5. Порядок виконання

1. Реалізувати арифметику великих чисел

2. Створити генератор простих чисел з тестом Міллера-Рабіна

3. Реалізувати алгоритм генерації ключових пар

4. Створити функції шифрування та дешифрування

5. Додати оптимізацію через китайську теорему про залишки

6. Реалізувати схему доповнення PKCS#1

7. Створити модуль цифрового підпису

8. Провести тестування та аналіз безпеки

6. Результат

Програма повинна надавати можливість генерувати ключові пари різних розмірів, шифрувати та дешифрувати повідомлення, створювати та перевіряти цифрові підписи. Необхідно продемонструвати різницю в швидкості між базовою реалізацією та оптимізованою через CRT, показати важливість використання схем доповнення. Результати повинні включати графіки залежності часу операцій від розміру ключа та аналіз стійкості до відомих атак.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Модуль арифметики великих чисел

- Функція генерації простих чисел

- Функція генерації ключової пари RSA

- Функції шифрування та дешифрування

- Реалізація оптимізації CRT
- Модуль схеми доповнення PKCS#1
- Модуль цифрового підпису

2. Продемонструвати роботу додатку:
- Генерація ключових пар різних розмірів
- Шифрування та дешифрування тестових повідомлень
- Створення та перевірка цифрових підписів
- Порівняння швидкості з/без оптимізації CRT
- Демонстрація важливості правильного доповнення

3. Бути готовим відповісти на питання:
- Чому важливо генерувати прості числа приблизно однакового розміру?
- Як захистити реалізацію від атак по сторонніх каналах?
- Які існують атаки на RSA при неправильному використанні?
- Як вибрати безпечний розмір ключа для конкретного застосування?
- Чому не можна використовувати малі значення e або d?

Тести

1. Функція Ейлера φ(pq) для простих p та q дорівнює:

1) pq

2) (p-1)(q-1)

3) p + q

4) pq - 1

2. Стандартне значення відкритої експоненти e в RSA:

1) 3

2) 17

3) 65537

4) 2^16

3. Для прискорення дешифрування RSA використовується:

1) Алгоритм Евкліда

2) Китайська теорема про залишки

3) Мала теорема Ферма

4) Решето Ератосфена

4. Схема OAEP забезпечує:

1) Стиснення даних

2) Прискорення шифрування

3) Захист від адаптивних атак

4) Зменшення розміру ключа

5. Безпека RSA базується на складності:

1) Дискретного логарифмування

2) Факторизації великих чисел

3) Вирішення систем рівнянь

4) Пошуку колізій

Відповіді:

1-2,

2-3,

3-2,

4-3,

5-2

Тести підвищеної складності

1. При використанні e = 3 та відсутності доповнення можлива атака:

1) Вінера

2) Кубічного кореня

3) Циклічна

4) Факторизації

2. Атака Вінера ефективна коли:

1) e мале

2) d < n^0.25

3) p та q близькі

4) n парне

3. Для захисту від атаки Белмаре p та q повинні:

1) Бути простими Софі Жермен

2) Мати велике НСД(p-1, q-1)

3) Відрізнятися не менше ніж на 2^(n/2-100)

4) Бути близнюками

4. Multiprime RSA використовує:

1) Кілька пар ключів

2) $n = p_1p_2...p_k$ для $k > 2$

3) Кілька значень e

4) Додаткове шифрування

5. Проблема малих розв'язків для RSA полягає в:

1) Можливості знайти m при малому e

2) Вразливості до перебору

3) Повільному шифруванні

4) Великому розмірі ключів

6. CRT-RSA вразливий до атаки:

1) По сторонніх каналах при помилках

2) Грубої сили

3) Днів народження

4) Зустріч посередині

7. Сліпий підпис RSA використовується для:

1) Прискорення підписування

2) Анонімних транзакцій

3) Групового підпису

4) Шифрування ключів

8. RSA-PSS відрізняється від PKCS#1 v1.5:

1) Швидкістю

2) Довжиною підпису

3) Ймовірнісним доповненням

4) Алгоритмом хешування

9. Broadcast attack на RSA можлива при:

1) Однаковому m, різних (n,e), малому e

2) Великому e

3) Використанні CRT

4) Малому n

10. Для post-quantum безпеки RSA потребує ключів:

1) 2048 біт

2) 4096 біт

3) 8192 біт

4) Неможливо досягти

Відповіді:

1-b Без доповнення при e=3 можна витягти кубічний корінь

2-b Атака використовує розклад e/d в ланцюговий дріб

3-c Захист від факторизації методом Ферма

4-b Використання більше двох простих множників

5-a При малому m та e можливо m^e < n

6-a Помилка в CRT може розкрити фактори n

7-b Дозволяє підписувати не знаючи змісту

8-c PSS використовує випадкову сіль

9-a Китайська теорема дозволяє відновити m

10-d RSA принципово вразливий до квантових атак

Завдання для самостійної роботи

1. Реалізуйте атаку факторизації методом Ферма для RSA з близькими p та q. Визначте безпечну різницю між простими числами.

2. Створіть програму для демонстрації атаки кубічного кореня при e=3 без використання доповнення.

3. Реалізуйте схему порогового RSA, де для дешифрування потрібна кооперація k з n учасників.

4. Дослідіть вплив використання малих значень d на безпеку RSA та реалізуйте атаку Вінера.

5. Створіть систему сліпих підписів RSA для електронного голосування.

6. Реалізуйте RSA-OAEP з різними хеш-функціями та порівняйте безпеку.

7. Дослідіть атаки по сторонніх каналах на RSA та реалізуйте захищену версію.

8. Створіть multiprime RSA та порівняйте продуктивність з класичним RSA.

9. Реалізуйте гомоморфні властивості RSA для обчислень над зашифрованими даними.

10. Розробіть систему розподіленої генерації RSA ключів без довіреної сторони.

Теми лабораторних робіт

1. Базова реалізація RSA та дослідження впливу параметрів на безпеку

2. Оптимізація RSA з використанням китайської теореми про залишки

3. Реалізація та порівняння схем доповнення PKCS#1 та OAEP

4. Система цифрових підписів на основі RSA

5. Атаки на RSA при неправильному використанні

6. Гібридні криптосистеми на основі RSA

7. Пороговий RSA для розподілених систем

8. Сліпі підписи RSA та їх застосування

9. Захист RSA від атак по сторонніх каналах

10. Дослідження продуктивності RSA для різних застосувань

Алгоритм Діффі-Геллмана

Теоретичні відомості

Алгоритм Діффі-Геллмана є одним з найважливіших криптографічних протоколів, який дозволяє двом сторонам встановити спільний секретний ключ через незахищений канал зв’язку. Цей алгоритм був розроблений Вітфілдом Діффі та Мартіном Геллманом у 1976 році і став революційним проривом у криптографії, оскільки вперше дозволив вирішити проблему обміну ключами без попередньої домовленості.

Основна ідея алгоритму базується на складності обчислення дискретного логарифма в скінченному полі. Це означає, що хоча піднесення числа до степеня за модулем є обчислювально простою операцією, зворотна операція - знаходження показника степеня, знаючи основу та результат - є надзвичайно складною для великих чисел.

Алгоритм використовує властивості модульної арифметики та циклічних груп. Нехай p - велике просте число, а g - примітивний корінь за модулем p. Примітивний корінь - це таке число, степені якого за модулем p генерують усі числа від 1 до p-1.

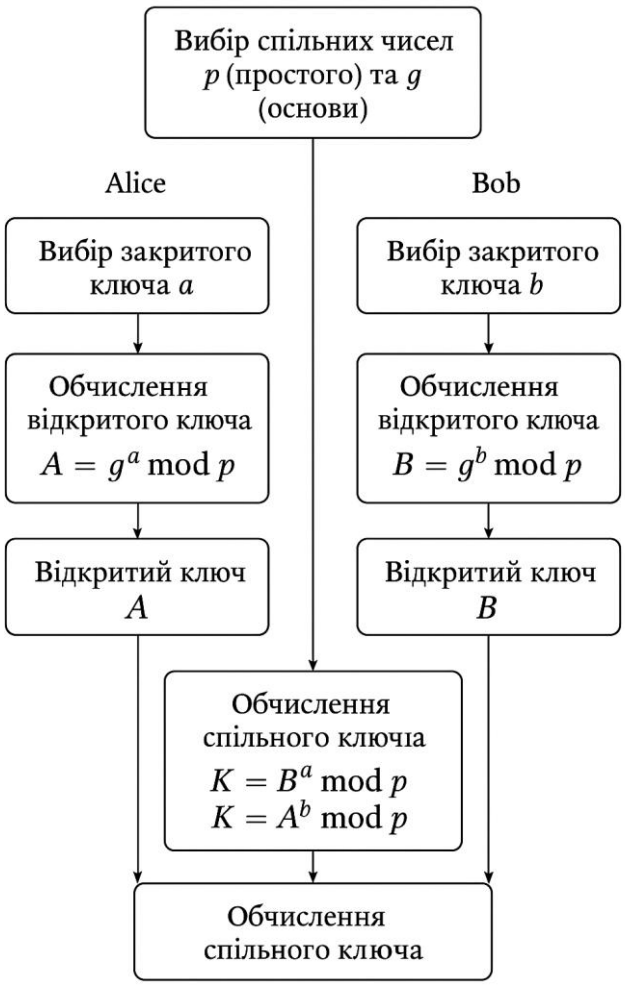
Для будь-яких цілих чисел a та b виконується властивість:

$$(g\text{^}a \bmod p)\text{^}b \bmod p = (g\text{^}b \bmod p)\text{^}a \bmod p = g\text{^}(ab) \bmod p$$

Саме ця властивість лежить в основі протоколу обміну ключами.

Опис алгоритму

Розглянемо детальний алгоритм роботи протоколу Діффі-Геллмана для двох учасників, яких традиційно називають Аліса та Боб.



На цій приведено блок-діаграму для ілюстрації алгоритму Діффі–Геллмана.

Розглянемо детальніше алгоритм:

Етап підготовки:

Спочатку обидва учасники домовляються про спільні параметри:

- p - велике просте число (зазвичай не менше 2048 біт для забезпечення належного рівня безпеки)

- g - примітивний корінь за модулем p

Ці параметри можуть бути відкритими і відомими всім, включаючи потенційних зловмисників.

Етап генерації ключів:

Аліса виконує наступні дії:

- Генерує випадкове секретне число a, де 1 < a < p-1

- Обчислює A = g^a mod p

- Надсилає A Бобу через відкритий канал

Боб виконує аналогічні дії:

- Генерує випадкове секретне число b, де 1 < b < p-1

- Обчислює B = g^b mod p

- Надсилає B Алісі через відкритий канал

Етап обчислення спільного секрету:

Аліса, отримавши B від Боба:

- Обчислює K = B^a mod p = ($g^{b)}$a mod p = g^(ab) mod p

Боб, отримавши A від Аліси:

- Обчислює K = A^b mod p = ($g^{a)}$b mod p = g^(ab) mod p

Таким чином, обидва учасники отримують однаковий секретний ключ

K = g^(ab) mod p,

не передаючи його через канал зв’язку.

Наведемо псевдокод для реалізації алгоритму Діффі-Геллмана:

```
Функція modPow(основа, показник, модуль):
    результат = 1
    основа = основа mod модуль
    поки показник > 0:
        якщо показник mod 2 == 1:
            результат = (результат * основа) mod модуль
        показник = показник >> 1
        основа = (основа * основа) mod модуль
    повернути результат

Функція генеруватиПриватнийКлюч(p):
```

```
    повернути випадковеЧисло(2, p-2)

Функція обчислитиПублічнийКлюч(g, приватнийКлюч, p):
    повернути modPow(g, приватнийКлюч, p)

Функція обчислитиСпільнийСекрет(публічнийКлючПартнера, приватнийКлюч,
p):
    повернути modPow(публічнийКлючПартнера, приватнийКлюч, p)

Алгоритм ДіффіГеллмана(p, g):
    // Для Аліси
    a = генеруватиПриватнийКлюч(p)
    A = обчислитиПублічнийКлюч(g, a, p)

    // Для Боба
    b = генеруватиПриватнийКлюч(p)
    B = обчислитиПублічнийКлюч(g, b, p)

    // Обмін публічними ключами
    // Аліса надсилає A Бобу
    // Боб надсилає B Алісі

    // Обчислення спільного секрету
    K_аліса = обчислитиСпільнийСекрет(B, a, p)
    K_боб = обчислитиСпільнийСекрет(A, b, p)

    // K_аліса == K_боб
    повернути K_аліса
```

Розглянемо роботу алгоритму на конкретному числовому прикладі. Для демонстрації використаємо невеликі числа, хоча в реальних застосуваннях використовуються числа розміром у тисячі біт.

Крок 1: Вибір спільних параметрів

Нехай p = 23 (просте число) та g = 5 (примітивний корінь за модулем 23).

Перевіримо, що g = 5 дійсно є примітивним коренем за модулем 23:

- 5^1 mod 23 = 5 - 5^2 mod 23 = 2 - 5^3 mod 23 = 10 - 5^4 mod 23 = 4

-

...

і так далі

Степені числа 5 генерують усі числа від 1 до 22, що підтверджує його властивість примітивного кореня.

Крок 2: Генерація приватних ключів

Аліса вибирає секретне число a = 6 Боб вибирає секретне число b = 15

Крок 3: Обчислення публічних ключів

Аліса обчислює:

A = g^a mod p = 5^6 mod 23 = 15625 mod 23 = 8

Боб обчислює:

B = g^b mod p = 5^15 mod 23 = 30517578125 mod 23 = 19

Крок 4: Обмін публічними ключами

Аліса надсилає число 8 Бобу через відкритий канал. Боб надсилає число 19 Алісі через відкритий канал.

Крок 5: Обчислення спільного секрету

Аліса обчислює:

K = B^a mod p = 19^6 mod 23 = 47045881 mod 23 = 2

Боб обчислює:

K = A^b mod p = 8^15 mod 23 = 35184372088832 mod 23 = 2

Обидва учасники отримали однаковий секретний ключ K = 2, не передаючи його через канал зв'язку.

Безпека алгоритму Діффі-Геллмана базується на складності розв'язання задачі дискретного логарифмування. Зловмисник, який перехоплює публічні значення A та B, а також знає параметри p та g, повинен розв'язати одне з рівнянь:

- A = g^a mod p (знайти a)

- B = g^b mod p (знайти b)

Для великих значень p ця задача є обчислювально нерозв'язною за розумний час навіть з використанням найпотужніших комп'ютерів.

Область застосування

Алгоритм Діффі-Геллмана широко застосовується в сучасній криптографії та інформаційній безпеці:

- Протоколи безпечного зв'язку: TLS/SSL використовують варіант алгоритму Діффі-Геллмана для встановлення безпечного з'єднання між веб-браузером та сервером.

- VPN-з’єднання: Багато VPN-протоколів, включаючи IPSec та OpenVPN, використовують алгоритм для встановлення безпечного тунелю між клієнтом та сервером.

- Месенджери з наскрізним шифруванням: Signal, WhatsApp та інші безпечні месенджери використовують варіанти алгоритму для обміну ключами шифрування між користувачами.

- SSH-протокол: Secure Shell використовує алгоритм Діффі-Геллмана для встановлення безпечного з’єднання з віддаленим сервером.

- Криптовалюти та блокчейн: Деякі криптографічні протоколи в блокчейн-системах використовують принципи алгоритму для забезпечення конфіденційності транзакцій.

Приклади застосування

Приклад 1: Безпечний обмін повідомленнями

Розглянемо сценарій, де два користувачі хочуть обмінюватися зашифрованими повідомленнями через публічну мережу. Вони можуть використати алгоритм Діффі-Геллмана наступним чином:

Користувач А та користувач Б домовляються про параметри p = 2^2048 - 2^1984 - 1 + 2^64 * (⌊2^1918 * π⌋ + 124476) та відповідний генератор g. Кожен генерує свій приватний ключ, обчислює публічний ключ та обмінюється ним з партнером. Після обчислення спільного секрету вони можуть використовувати його як ключ для симетричного алгоритму шифрування, наприклад AES-256.

Приклад 2: Встановлення HTTPS-з’єднання

Коли користувач заходить на захищений веб-сайт, браузер та сервер використовують варіант алгоритму Діффі-Геллмана під назвою Ephemeral Diffie-Hellman (DHE або ECDHE для еліптичних кривих). Процес відбувається наступним чином:

Сервер генерує тимчасову пару ключів для кожного з’єднання, підписує публічний ключ своїм сертифікатом та надсилає клієнту. Клієнт перевіряє підпис, генерує свою пару ключів та надсилає публічний ключ серверу. Обидві сторони обчислюють спільний секрет, який використовується для генерації сесійних ключів шифрування.

Приклад 3: Груповий обмін ключами

Алгоритм може бути розширений для роботи з більш ніж двома учасниками. Наприклад, для трьох учасників процес виглядає так:

Учасники А, Б та В генерують приватні ключі a, b та c відповідно. На першому етапі кожен обчислює та надсилає іншим g^x mod p, де x - їхній приватний ключ. На другому етапі кожен обчислює (g^y)x mod p для отриманого значення та надсилає результат третьому учаснику. На третьому етапі всі обчислюють спільний ключ K = g^(abc) mod p.

Переваги та недоліки

Переваги

- Відсутність необхідності в попередньому розподілі ключів: Учасники можуть встановити спільний секрет без попередньої зустрічі чи обміну секретною інформацією.

- Простота реалізації: Алгоритм використовує лише базові арифметичні операції - множення та піднесення до степеня за модулем.

- Масштабованість безпеки: Рівень безпеки можна легко підвищити, збільшивши розмір використовуваних чисел.

- Математична елегантність: Алгоритм базується на фундаментальних математичних принципах теорії чисел.

- Широка підтримка: Алгоритм реалізований у більшості криптографічних бібліотек та підтримується стандартами.

Недоліки

- Вразливість до атаки “людина посередині”: Без додаткової автентифікації зловмисник може виступати посередником між учасниками.

- Обчислювальна складність для великих чисел: Операції з числами розміром у тисячі біт вимагають значних обчислювальних ресурсів.

- Необхідність у безпечній генерації випадкових чисел: Слабкі генератори випадкових чисел можуть скомпрометувати безпеку системи.

- Відсутність прямої підтримки автентифікації: Алгоритм забезпечує лише конфіденційність, але не автентичність учасників.

- Потенційна вразливість до квантових комп’ютерів: Алгоритм Шора може ефективно розв’язувати задачу дискретного логарифмування на квантових комп’ютерах.

Контрольні запитання

- На якій математичній проблемі базується безпека алгоритму Діффі-Геллмана?

- Чому важливо, щоб g був примітивним коренем за модулем p?

- Які дані передаються через відкритий канал під час виконання протоколу?

- Як можна захистити алгоритм від атаки “людина посередині”?

- Чому приватні ключі повинні бути випадковими?

- Яка роль модульної арифметики в алгоритмі?

- Як розмір простого числа p впливає на безпеку алгоритму?

- Чи можна використовувати один і той же приватний ключ для різних сеансів?

- Які існують варіанти алгоритму Діффі-Геллмана?

- Як алгоритм може бути адаптований для більш ніж двох учасників?

Методичні вказівки до лабораторної роботи

Назва

Реалізація та дослідження алгоритму Діффі-Геллмана для обміну ключами

Мета

Вивчити принципи роботи алгоритму Діффі-Геллмана, реалізувати його програмно та дослідити вплив параметрів на безпеку та продуктивність алгоритму.

Завдання

- Реалізувати функції для роботи з великими числами та модульної арифметики

- Реалізувати генерацію простих чисел заданого розміру

- Реалізувати пошук примітивного кореня для простого числа

- Реалізувати повний протокол обміну ключами Діффі-Геллмана

- Створити демонстраційну програму для візуалізації роботи алгоритму

- Дослідити вплив розміру параметрів на час виконання

- Реалізувати атаку “людина посередині” та продемонструвати її роботу

Опис

Лабораторна робота передбачає створення повноцінної реалізації алгоритму Діффі-Геллмана з усіма необхідними допоміжними функціями. Програма повинна підтримувати роботу з числами довільної розрядності, генерувати криптографічно стійкі параметри та демонструвати всі етапи протоколу.

Порядок виконання

- Створити модуль для роботи з великими числами або використати готову бібліотеку
- Реалізувати функцію швидкого піднесення до степеня за модулем
- Реалізувати тест Міллера-Рабіна для перевірки простоти чисел
- Реалізувати функцію генерації простих чисел заданої довжини
- Реалізувати функцію пошуку примітивного кореня
- Створити класи для учасників протоколу (Аліса та Боб)
- Реалізувати методи генерації ключів та обчислення спільного секрету
- Створити інтерфейс для демонстрації роботи алгоритму
- Провести тестування з різними розмірами параметрів
- Реалізувати та продемонструвати атаку “людина посередині”

Результат

Програма повинна демонструвати повний цикл роботи алгоритму Діффі-Геллмана, включаючи генерацію параметрів, обмін ключами та обчислення спільного секрету. Також повинна бути можливість спостерігати за всіма проміжними значеннями та часом виконання операцій.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
- Модуль для роботи з великими числами
- Функції модульної арифметики
- Функції генерації та перевірки простих чисел
- Функції пошуку примітивного кореня
- Класи учасників протоколу
- Демонстраційний інтерфейс
- Продемонструвати роботу додатку:

- Генерацію безпечних параметрів p та g

- Повний цикл обміну ключами між двома учасниками

- Перевірку того, що обидва учасники отримали однаковий ключ

- Роботу з параметрами різного розміру (512, 1024, 2048 біт)

- Демонстрацію атаки “людина посередині”

- Бути готовим відповісти на питання:

- Як працює алгоритм швидкого піднесення до степеня за модулем?

- Чому важливо використовувати криптографічно стійкі генератори випадкових чисел?

- Як перевірити, чи є число примітивним коренем?

- Які методи можна використати для захисту від атаки “людина посередині”?

- Як вибір параметрів впливає на безпеку алгоритму?

Тести

- Основна складність, на якій базується безпека алгоритму Діффі-Геллмана:

- 1) Факторизація великих чисел

- 2) Дискретне логарифмування

- 3) Еліптичні криві

- 4) Хеш-функції

- Які дані передаються через відкритий канал в алгоритмі Діффі-Геллмана?

- 1) Приватні ключі учасників

- 2) Спільний секретний ключ

- 3) Публічні ключі учасників

- 4) Хеш від приватних ключів

- Що таке примітивний корінь за модулем p?

- 1) Найменше просте число, менше за p

- 2) Число, степені якого генерують всі елементи групи

- 3) Квадратний корінь з p

- 4) Число, що ділить p-1

- Яка властивість модульної арифметики використовується в алгоритмі?

- 1) (a+b) mod p = (a mod p + b mod p) mod p

- 2) (a*b) mod p = (a mod p b mod p) mod p

- 3) $(g^{a)}b \bmod p = (g^{b)}a \bmod p$

- 4) a^p mod p = a mod p

- До якої атаки вразливий базовий алгоритм Діффі-Геллмана?

- 1) Атака грубої сили

- 2) Атака “людина посередині”

- 3) Атака днів народження

- 4) Диференційний криптоаналіз

Відповіді:

1-2,

2-3,

3-2,

4-3,

5-2

Тести підвищеної складності

- Який мінімальний рекомендований розмір модуля p для забезпечення довгострокової безпеки?

- 1) 512 біт

- 2) 1024 біти

- 3) 2048 біт

- 4) 4096 біт

- Яка складність найкращого відомого алгоритму для розв’язання задачі дискретного логарифмування?

- 1) O(n)

- 2) O(n log n)

- 3) $O(n^2)$

- 4) O(e^((ln n)^(1/3) * (ln ln n)^(2/3)))

- Що таке Ephemeral Diffie-Hellman?

- 1) Версія з фіксованими параметрами

- 2) Версія з новими ключами для кожного сеансу

- 3) Версія для квантових комп’ютерів

- 4) Версія з подвійним шифруванням

- Яка кількість операцій потрібна для обчислення g^a mod p методом швидкого піднесення до степеня?

- 1) O(a)

- 2) O(log a)

- 3) O(√a)

- 4) O(a^2)

- Який варіант алгоритму використовує еліптичні криві?

- 1) DHE

- 2) DH-RSA

- 3) ECDHE

- 4) DH-AES

- Скільки спільних секретів можна встановити між n учасниками попарно?

- 1) n

- 2) n-1

- 3) n(n-1)/2

- 4) n^2

- Яка властивість повинна виконуватись для безпечного простого числа p?

- 1) p = 2q + 1, де q - також просте

- 2) p - 1 має багато малих дільників

- 3) p є числом Мерсенна

- 4) p є паліндромом

- Що відбудеться, якщо два учасники використають однакові приватні ключі?

- 1) Протокол не працюватиме

- 2) Спільний секрет буде g^(a^2) mod p

- 3) Безпека буде скомпрометована

- 4) Нічого особливого

- Яка атака може бути успішною, якщо використовується слабкий генератор випадкових чисел?

- 1) Атака на основі малих підгруп

- 2) Передбачення приватного ключа

- 3) Атака Полларда

- 4) Квантова атака

- Як називається проблема знаходження g^(ab) mod p, знаючи g^a mod p та g^b mod p?

- 1) Проблема дискретного логарифма

- 2) Проблема Діффі-Геллмана

- 3) Проблема факторизації

- 4) Проблема ізогенії

Відповіді:

1-3. Для довгострокової безпеки рекомендується використовувати модуль розміром щонайменше 2048 біт

2-4. Алгоритм числового поля має субекспоненційну складність

3-2. Ephemeral DH використовує нові ключі для кожного сеансу зв'язку

4-2. Швидке піднесення до степеня вимагає O(log a) множень

5-3. ECDHE використовує еліптичні криві замість мультиплікативної групи

6-3. Кількість попарних з'єднань між n учасниками дорівнює C(n,2) = n(n-1)/2

7-1. Безпечне просте число має вигляд p = 2q + 1, де q також просте

8-2. Якщо a = b, то спільний секрет буде g^(a*a) = g^(a^2) mod p

9-2. Слабкий генератор дозволяє передбачити "випадкові" приватні ключі

10-2. Це обчислювальна проблема Діффі-Геллмана (CDH)

Завдання для самостійної роботи

- Реалізуйте генератор безпечних простих чисел (p = 2q + 1) заданого розміру та порівняйте час генерації зі звичайними простими числами.

- Напишіть програму для пошуку всіх примітивних коренів для заданого простого числа та проаналізуйте їх розподіл.

- Реалізуйте атаку Полларда для дискретного логарифмування та визначте максимальний розмір модуля, для якого атака працює за прийнятний час.

- Створіть візуалізацію процесу обміну ключами з покроковою демонстрацією всіх обчислень та переданих даних.

- Реалізуйте розширену версію алгоритму для трьох та більше учасників і порівняйте складність з попарним обміном.

- Дослідіть вплив вибору різних примітивних коренів на розподіл публічних ключів та час обчислень.

- Реалізуйте захист від атаки “людина посередині” використовуючи цифрові підписи та порівняйте продуктивність.

- Створіть бенчмарк для порівняння швидкості різних реалізацій модульного піднесення до степеня.

- Реалізуйте версію алгоритму з використанням еліптичних кривих (ECDH) та порівняйте з класичною версією.

- Напишіть програму для статистичного аналізу розподілу спільних секретів при різних параметрах алгоритму.

Теми лабораторних робіт

- Реалізація базового протоколу Діффі-Геллмана з візуалізацією всіх етапів обміну ключами.

- Розробка системи безпечного обміну повідомленнями на основі алгоритму Діффі-Геллмана та симетричного шифрування.

- Створення генератора криптографічно стійких параметрів для алгоритму Діффі-Геллмана з різними рівнями безпеки.

- Реалізація та аналіз атаки “людина посередині” на протокол Діффі-Геллмана.

- Розробка групового протоколу обміну ключами на основі алгоритму Діффі-Геллмана.

- Порівняльний аналіз продуктивності алгоритму Діффі-Геллмана з різними розмірами параметрів.

- Реалізація протоколу Station-to-Station з використанням алгоритму Діффі-Геллмана та цифрових підписів.

- Створення бібліотеки для роботи з алгоритмом Діффі-Геллмана з підтримкою різних груп.

- Розробка навчального симулятора для демонстрації криптографічних атак на алгоритм Діффі-Геллмана.

- Реалізація гібридної криптосистеми з використанням алгоритму Діффі-Геллмана для обміну ключами.

Криптографія на еліптичних кривих (ECDH, ECDSA)

Теоретичні відомості

Криптографія на еліптичних кривих є одним з найсучасніших та найефективніших підходів до асиметричної криптографії. Вона базується на алгебраїчній структурі еліптичних кривих над скінченними полями та забезпечує той самий рівень безпеки, що й традиційні криптосистеми, але з використанням значно менших ключів.

Еліптична крива над полем дійсних чисел визначається рівнянням Вейєрштрасса:

$$y^2 = x^3 + ax + b$$

де a та b - константи, що задовольняють умову

$$4a^3 + 27b^2 \neq 0.$$

Ця умова гарантує, що крива не має особливих точок (самоперетинів або касп).

Для криптографічних застосувань використовуються еліптичні криві над скінченними полями. Найпоширенішими є криві над простим полем Fp, де p - велике просте число. У цьому випадку рівняння кривої має вигляд:

$$y^2 \equiv x^3 + ax + b \pmod p$$

Точками еліптичної кривої E(Fp) є всі пари (x, y), де x, y $\in$ Fp, що задовольняють рівняння кривої, плюс спеціальна точка O, яка називається точкою на нескінченності та виконує роль нейтрального елемента.

Операції над точками еліптичної кривої

Основною операцією на еліптичній кривій є додавання точок. Ця операція має геометричну інтерпретацію та алгебраїчне визначення.

Додавання точок:

Для двох різних точок $P = (x_1, y_1)$ та $Q = (x_2, y_2)$ на кривій, їх сума

$$R = P + Q = (x_3, y_3)$$

обчислюється за формулами:

Якщо $x_1 \neq x_2$:

- $\lambda = (y_2 - y_1) / (x_2 - x_1) \bmod p$
- $x_3 = \lambda^2 - x_1 - x_2 \bmod p$
- $y_3 = \lambda(x_1 - x_3) - y_1 \bmod p$

Подвоєння точки:

Для точки $P = (x_1, y_1)$, точка $2P = P + P = (x_3, y_3)$ обчислюється:

Якщо $y_1 \neq 0$:

- $\lambda = (3x_1^2 + a) / (2y_1) \bmod p$
- $x_3 = \lambda^2 - 2x_1 \bmod p$
- $y_3 = \lambda(x_1 - x_3) - y_1 \bmod p$

Скалярне множення:

Операція kP, де k - ціле число, а P - точка на кривій, визначається як додавання точки P до себе k разів: $kP = P + P + \ldots + P$ (k разів)

Ефективне обчислення скалярного множення виконується методом подвоєння та додавання, аналогічним швидкому піднесенню до степеня.

Наведемо псевдокод для основних операцій над точками еліптичної кривої:

```
Функція модульнеОбернення(a, p):
    // Обчислення a^(-1) mod p використовуючи розширений алгоритм Евк
ліда
    якщо a < 0:
        a = (a mod p + p) mod p
    g, x, _ = розширенийЕвклід(a, p)
    якщо g != 1:
        повернути помилку "Обернення не існує"
    повернути x mod p

Функція додаванняТочок(P, Q, a, p):
    // P та Q - точки на кривій y² = x³ + ax + b mod p
    якщо P == O:
        повернути Q
    якщо Q == O:
        повернути P

    x1, y1 = P
    x2, y2 = Q
```

```
    якщо x1 == x2:
        якщо y1 == y2:
            // Подвоєння точки
            λ = (3 * x1² + a) * модульнеОбернення(2 * y1, p) mod p
        інакше:
            // P + (-P) = O
            повернути O
    інакше:
        // Додавання різних точок
        λ = (y2 - y1) * модульнеОбернення(x2 - x1, p) mod p

    x3 = (λ² - x1 - x2) mod p
    y3 = (λ * (x1 - x3) - y1) mod p

    повернути (x3, y3)

Функція скалярнеМноження(k, P, a, p):
    // Обчислення kP методом подвоєння та додавання
    якщо k == 0:
        повернути O
    якщо k == 1:
        повернути P
    якщо k < 0:
        k = -k
        P = (P[0], -P[1] mod p)  // -P = (x, -y)

    результат = O
    адденд = P

    поки k > 0:
        якщо k mod 2 == 1:
            результат = додаванняТочок(результат, адденд, a, p)
        адденд = додаванняТочок(адденд, адденд, a, p)
        k = k // 2

    повернути результат
```

Алгоритм ECDH (Elliptic Curve Diffie-Hellman)

ECDH є аналогом класичного алгоритму Діффі-Геллмана, але використовує операції над точками еліптичної кривої замість операцій у мультиплікативній групі.

Параметри домену:

- Еліптична крива E, задана рівнянням $y^2 = x^3 + ax + b \mod p$

- Базова точка G порядку n на кривій

- Порядок n - просте число

Алгоритм обміну ключами:

Розглянемо детальний алгоритм ECDH:

```
Алгоритм ECDH(E, G, n):
    // Аліса:
    // 1. Генерує приватний ключ
    dA = випадковеЧисло(1, n-1)

    // 2. Обчислює публічний ключ
    QA = скалярнеМноження(dA, G, a, p)

    // 3. Надсилає QA Бобу

    // Боб:
    // 1. Генерує приватний ключ
    dB = випадковеЧисло(1, n-1)

    // 2. Обчислює публічний ключ
    QB = скалярнеМноження(dB, G, a, p)

    // 3. Надсилає QB Алісі

    // Обчислення спільного секрету:
    // Аліса обчислює:
    KA = скалярнеМноження(dA, QB, a, p)

    // Боб обчислює:
    KB = скалярнеМноження(dB, QA, a, p)

    // KA = KB = dA * dB * G
    спільнийСекрет = x-координата(KA)

    повернути спільнийСекрет
```

Алгоритм ECDSA (Elliptic Curve Digital Signature Algorithm)

ECDSA використовується для створення та перевірки цифрових підписів. Алгоритм забезпечує автентичність, цілісність та неспростовність повідомлень.

Генерація ключів:

```
Функція генераціяКлючівECDSA(E, G, n):
    // Генерація приватного ключа
    d = випадковеЧисло(1, n-1)

    // Обчислення публічного ключа
    Q = скалярнеМноження(d, G, a, p)

    повернути (d, Q)  // приватний та публічний ключі
```

Створення підпису:

```
Функція підписатиECDSA(повідомлення, d, E, G, n):
    // d - приватний ключ
    // Обчислення хешу повідомлення
    e = SHA256(повідомлення)
    z = найлівішіБіти(e, розмірБітів(n))

    повторювати:
        // Генерація випадкового числа k
        k = випадковеЧисло(1, n-1)

        // Обчислення точки kG
        (x1, y1) = скалярнеМноження(k, G, a, p)

        // Обчислення r
        r = x1 mod n

        якщо r == 0:
            продовжити

        // Обчислення s
        k_inv = модульнеОбернення(k, n)
        s = k_inv * (z + r * d) mod n

        якщо s != 0:
            вийти з циклу

    повернути (r, s)
```

Перевірка підпису:

```
Функція перевіритиПідписECDSA(повідомлення, підпис, Q, E, G, n):
    // Q - публічний ключ
    // підпис = (r, s)
    r, s = підпис

    // Перевірка діапазону значень
    якщо r < 1 або r >= n або s < 1 або s >= n:
```

```
        повернути false

    // Обчислення хешу повідомлення
    e = SHA256(повідомлення)
    z = найлівішіБіти(e, розмірБітів(n))

    // Обчислення w = s^(-1) mod n
    w = модульнеОбернення(s, n)

    // Обчислення u1 та u2
    u1 = z * w mod n
    u2 = r * w mod n

    // Обчислення точки (x1, y1) = u1*G + u2*Q
    точка1 = скалярнеМноження(u1, G, a, p)
    точка2 = скалярнеМноження(u2, Q, a, p)
    (x1, y1) = додаванняТочок(точка1, точка2, a, p)

    // Перевірка підпису
    якщо r == x1 mod n:
        повернути true
    інакше:
        повернути false
```

Розглянемо роботу алгоритмів ECDH та ECDSA на конкретному прикладі з невеликими числами для наочності.

Параметри еліптичної кривої:

- Крива: $y^2 = x^3 + 2x + 3 \bmod 97$

- Базова точка: G = (3, 6)

- Порядок базової точки: n = 5

Перевіримо, що G дійсно лежить на кривій:

$6^2 \bmod 97 = 36$ $3^3 + 2 \cdot 3 + 3 \bmod 97 = 27 + 6 + 3 = 36$

Отже, точка G = (3, 6) належить кривій.

Приклад роботи ECDH:

Крок 1: Генерація ключів

Аліса вибирає приватний ключ dA = 2 QA = 2G = G + G

Обчислимо 2G використовуючи формулу подвоєння:

$\lambda = (3 \cdot 3^2 + 2) / (2 \cdot 6) \bmod 97 = 29 / 12 \bmod 97$

Знайдемо

12^(-1) mod 97: 12^(-1) mod 97 = 89 (оскільки 12·89 = 1068 = 11·97 + 1)

λ = 29·89 mod 97 = 2581 mod 97 = 60

$x_з = 60^2 - 2·3 \mod 97 = 3600 - 6 \mod 97 = 14$ $y_з = 60·(3 - 14) - 6 \mod 97 = -660 - 6 \mod 97 = 19$

Отже, QA = (14, 19)

Боб вибирає приватний ключ:

dB = 3 QB = 3G = 2G + G = (14, 19) + (3, 6)

Обчислимо QB:

λ = (6 - 19) / (3 - 14) mod 97 = -13 / -11 mod 97 = 84 / 86 mod 97

86^(-1) mod 97 = 45 λ = 84·45 mod 97 = 3780 mod 97 = 93

$x_з = 93^2 - 14 - 3 \mod 97 = 8649 - 17 \mod 97 = 11$ $y_з = 93·(14 - 11) - 19 \mod 97 = 279 - 19 \mod 97 = 66$

Отже, QB = (11, 66)

Крок 2: Обчислення спільного секрету

Аліса обчислює:

KA = dA·QB = 2·(11, 66)

Боб обчислює:

KB = dB·QA = 3·(14, 19)

Обидва отримають точку 6G = (38, 90), і спільний секрет буде x-координатою: K = 38

Приклад роботи ECDSA:

Використаємо ті ж параметри кривої для демонстрації ECDSA.

Генерація ключів:

Приватний ключ: d = 3

Публічний ключ: Q = 3G = (11, 66)

Підписання повідомлення:

Нехай хеш повідомлення z = 4.

Вибираємо k = 2

Обчислюємо

$(x_1, y_1) = 2G = (14, 19)$ $r = 14 \mod 5 = 4$

Обчислюємо s:

k^(-1) mod 5 = 2^(-1) mod 5 = 3 (оскільки 2·3 = 6 = 1 mod 5) s = 3·(4 + 4·3) mod 5 = 3·16 mod 5 = 48 mod 5 = 3

Підпис:

(r, s) = (4, 3)

Перевірка підпису:

w = s^(-1) mod 5 = 3^(-1) mod 5 = 2 u_1 = 4·2 mod 5 = 3 u_2 = 4·2 mod 5 = 3

Обчислюємо

$u_1G + u_2Q = 3G + 3Q = 3G + 3 \cdot 3G = 3G + 9G = 12G = 2G = (14, 19)$

Перевіряємо:

r = 4 та x_1 mod 5 = 14 mod 5 = 4

Оскільки вони рівні, підпис дійсний.

Область застосування

Криптографія на еліптичних кривих широко застосовується в сучасних системах безпеки:

- TLS/SSL протоколи: Більшість сучасних веб-серверів підтримують ECDHE для обміну ключами, що забезпечує perfect forward secrecy. Популярні криві включають P-256, P-384 та Curve25519.

- Криптовалюти: Bitcoin використовує криву secp256k1 для генерації адрес та підписування транзакцій. Ethereum та багато інших криптовалют також використовують ECC.

- Мобільні пристрої: Через менший розмір ключів ECC ідеально підходить для пристроїв з обмеженими ресурсами. Apple використовує ECC в iOS для різних криптографічних операцій.

- Смарт-карти та IoT: Обмежені обчислювальні ресурси та пам’ять роблять ECC оптимальним вибором для вбудованих систем.

- Державні стандарти: NSA рекомендує Suite B криптографію, яка включає ECDH та ECDSA. Багато країн прийняли національні стандарти на основі ECC.

- Месенджери: Signal Protocol використовує Curve25519 для обміну ключами в WhatsApp, Signal та інших безпечних месенджерах.

Приклади застосування

Приклад 1: Безпечний обмін файлами

Розглянемо систему безпечного обміну файлами між двома користувачами. Система використовує ECDH для встановлення сесійного ключа та ECDSA для автентифікації.

Користувачі генерують довготривалі пари ключів ECDSA для ідентифікації. При кожному сеансі обміну файлами генеруються ephemeral ECDH ключі. Файл шифрується AES-256 з ключем, отриманим з ECDH спільного секрету через KDF. Зашифрований файл підписується ECDSA для забезпечення автентичності та цілісності.

Приклад 2: Автентифікація в IoT мережі

У мережі розумних пристроїв кожен пристрій має унікальну пару ключів ECDSA, згенеровану при виробництві. При підключенні до мережі пристрій підписує challenge від сервера своїм приватним ключем. Сервер перевіряє підпис використовуючи публічний ключ пристрою з бази даних. Після успішної автентифікації встановлюється захищений канал через ECDH.

Приклад 3: Blockchain транзакції

У системі Bitcoin кожна адреса гаманця походить від публічного ключа ECDSA. Для створення транзакції власник підписує її своїм приватним ключем. Мережа перевіряє підпис, використовуючи публічний ключ, який можна обчислити з адреси. Це забезпечує доказ володіння коштами без розкриття приватного ключа.

Переваги та недоліки

Переваги

- Менший розмір ключів: ECC забезпечує той самий рівень безпеки з ключами в 3-6 разів меншими, ніж RSA. Наприклад, 256-бітний ключ ECC еквівалентний 3072-бітному ключу RSA.

- Вища швидкість: Операції з меншими числами виконуються швидше, особливо на пристроях з обмеженими ресурсами.

- Менше використання пам’яті та пропускної здатності: Менші ключі та підписи економлять місце в сховищах та трафік при передачі.

- Кращі алгоритмічні властивості: Складність найкращих відомих атак на ECC експоненційна, на відміну від субекспоненційних атак на RSA.

- Гнучкість параметрів: Можна вибирати різні криві для різних рівнів безпеки та оптимізації.

Недоліки

- Складність реалізації: Правильна та безпечна реалізація ECC вимагає глибокого розуміння математики еліптичних кривих.

- Патентні обмеження: Деякі оптимізації та криві були захищені патентами, хоча більшість вже закінчилися.

- Вразливість до неправильного вибору параметрів: Використання слабких кривих може скомпрометувати безпеку.

- Складність валідації параметрів: Перевірка того, що крива має необхідні криптографічні властивості, є нетривіальною задачею.

- Потенційні бекдори: Існують побоювання щодо навмисно ослаблених стандартних кривих, особливо NIST кривих.

Контрольні запитання

- Яка основна математична структура використовується в ECC?

- Чому умова $4a^3 + 27b^2 \neq 0$ важлива для еліптичної кривої?

- Як виконується додавання двох різних точок на еліптичній кривій?

- Що таке точка на нескінченності та яку роль вона відіграє?

- Як співвідносяться розміри ключів ECC та RSA для однакового рівня безпеки?

- Яка основна відмінність між ECDH та класичним алгоритмом Діффі-Геллмана?

- Чому важливо використовувати випадкове k при кожному підписанні в ECDSA?

- Що таке порядок точки на еліптичній кривій?

- Які переваги дає використання ephemeral ключів в ECDH?

- Як перевірити, чи належить точка еліптичній кривій?

Методичні вказівки до лабораторної роботи

Назва

Реалізація та дослідження криптографічних алгоритмів на еліптичних кривих ECDH та ECDSA

Мета

Вивчити математичні основи еліптичних кривих, реалізувати алгоритми ECDH та ECDSA, дослідити їх властивості та порівняти з класичними криптосистемами.

Завдання

- Реалізувати арифметику еліптичних кривих над простим полем
- Реалізувати алгоритм ECDH для обміну ключами
- Реалізувати алгоритм ECDSA для цифрового підпису
- Створити програму для генерації безпечних параметрів кривої
- Провести порівняльний аналіз продуктивності з RSA
- Реалізувати атаку на ECDSA при повторному використанні k
- Візуалізувати операції над точками еліптичної кривої

Опис

Лабораторна робота передбачає повну реалізацію криптосистеми на еліптичних кривих, включаючи всі базові операції, генерацію параметрів та криптографічні протоколи. Особлива увага приділяється правильній реалізації арифметики в скінченному полі та захисту від атак.

Порядок виконання

- Реалізувати клас для роботи з точками еліптичної кривої
- Реалізувати методи додавання точок та скалярного множення
- Створити функції для перевірки параметрів кривої
- Реалізувати генерацію ключових пар для ECDH та ECDSA
- Реалізувати протокол ECDH з повним циклом обміну
- Реалізувати створення та перевірку підписів ECDSA
- Створити тестові сценарії для різних кривих
- Реалізувати візуалізацію для малих кривих
- Провести аналіз продуктивності
- Продемонструвати вразливості при неправильному використанні

Результат

Готова криптографічна система на еліптичних кривих з можливістю обміну ключами через ECDH та створення цифрових підписів через ECDSA, включаючи візуалізацію та аналіз безпеки.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:

- Клас EllipticCurve для представлення кривої

- Клас Point для роботи з точками

- Модуль field_arithmetic для операцій у скінченному полі

- Модуль ecdh для протоколу обміну ключами

- Модуль ecdsa для роботи з підписами

- Модуль curve_parameters для стандартних кривих

- Продемонструвати роботу додатку:

- Генерацію та валідацію параметрів кривої

- Повний цикл ECDH між двома сторонами

- Створення та успішну перевірку підпису ECDSA

- Виявлення неправильного підпису

- Візуалізацію операцій для малої кривої

- Порівняння швидкості з RSA-2048

Бути готовим відповісти на питання:

- Як перевірити, що точка має заданий порядок?

- Чому важливо використовувати константний час для скалярного множення?

- Що станеться, якщо використати одне k для двох різних підписів?

- Як вибрати безпечну еліптичну криву?

- Які side-channel атаки можливі на ECC?

Тести

- Яка умова повинна виконуватися для параметрів a та b еліптичної кривої $y^2 = x^3 + ax + b$?

- 1) $a + b \neq 0$

- 2) $4a^3 + 27b^2 \neq 0$

- 3) $a^2 + b^2 = 1$

- 4) $ab \neq 0$

- Який елемент є нейтральним для операції додавання точок на еліптичній кривій?

- 1) Точка (0, 0)

- 2) Точка (1, 1)

- 3) Точка на нескінченності O

- 4) Базова точка G

- Що є спільним секретом в алгоритмі ECDH?

- 1) Сума приватних ключів

- 2) Добуток публічних ключів

- 3) x-координата точки dA·dB·G

- 4) y-координата базової точки

- Яке значення повинно бути унікальним для кожного підпису в ECDSA?

- 1) Приватний ключ d

- 2) Випадкове число k

- 3) Хеш повідомлення

- 4) Публічний ключ Q

- Який розмір ключа ECC приблизно еквівалентний 2048-бітному ключу RSA?

- 1) 112 біт

- 2) 224 біти

- 3) 384 біти

- 4) 521 біт

Відповіді:

1-2,

2-3,

3-3,

4-2,

5-2

Тести підвищеної складності

1. Що таке twist-атака на еліптичні криві?

- 1) Атака на слабкі генератори випадкових чисел
- 2) Використання точок на спорідненій кривій
- 3) Атака на малі підгрупи
- 4) Квантова атака

2. Яка властивість кривої гарантує стійкість до MOV-атаки?
- 1) Просте p повинно бути безпечним простим
- 2) Порядок n не ділить p^k - 1 для малих k
- 3) Крива повинна бути суперсингулярною
- 4) Дискримінант повинен бути квадратичним лишком

3. Що означає cofactor еліптичної кривої?
- 1) Кількість точок порядку 2
- 2) Відношення |E(Fp)| / n
- 3) Степінь розширення поля
- 4) Кількість автоморфізмів

4. Яка крива рекомендована для використання в сучасних системах?
- 1) NIST P-256
- 2) secp256k1
- 3) Curve25519
- 4) Всі перераховані

5. Що таке точка кручення на еліптичній кривій?
- 1) Точка з від’ємною y-координатою
- 2) Точка скінченного порядку
- 3) Точка на twist кривій
- 4) Сингулярна точка

6. Який метод використовується для захисту від timing атак при скалярному множенні?
- 1) Рандомізація координат
- 2) сходи Монтгомері
- 3) Подвійне обчислення
- 4) Використання якобіанових координат

7. Що таке степеня вкладення еліптичної кривої?

- 1) Розмірність простору кривої

- 2) Найменше k таке, що n | p^k - 1

- 3) Степінь мінімального полінома

- 4) Кількість точок зламу

8. Яка координатна система найефективніша для подвоєння точок?

- 1) Афінні координати

- 2) Проективні координати

- 3) Якобіанові координати

- 4) Координати Едвардса

9. Що перевіряється при валідації публічного ключа ECDH?

- 1) Тільки приналежність кривій

- 2) Приналежність кривій та nQ = O

- 3) Тільки nQ = O

- 4) Унікальність ключа

10. Яка атака можлива при детермінованій генерації k в ECDSA?

- 1) Відновлення приватного ключа

- 2) Підробка підпису

- 3) Атака на хеш-функцію

- 4) DoS атака

Відповіді:

1-b. Twist-атака використовує точки на спорідненій кривій з малим порядком

2-b. Стійкість до MOV вимагає великого степеня вкладення

3-b. Cofactor це відношення загальної кількості точок до порядку підгрупи

4-d. Всі три криві широко використовуються в різних застосуваннях

5-b. Точка кручення має скінченний порядок nP = O для деякого n

6-b. алгоритм сходи Монтгомері забезпечує константний час виконання

7-b. степеня вкладення визначає складність зведення ECDLP до DLP

8-c. Якобіанові координати оптимальні для подвоєння через відсутність ділень

9-b. Необхідно перевірити і приналежність кривій, і правильний порядок

10-a. Детерміноване k дозволяє відновити приватний ключ з двох підписів

Завдання для самостійної роботи

1. Реалізуйте арифметику еліптичних кривих над полем F2^m та порівняйте продуктивність з кривими над Fp.
2. Дослідіть та реалізуйте метод генерації кривих з комплексним множенням для створення безпечних параметрів.
3. Реалізуйте протокол ECIES (Elliptic Curve Integrated Encryption Scheme) для гібридного шифрування.
4. Створіть програму для факторизації чисел методом еліптичних кривих Ленстри.
5. Реалізуйте паралельну версію скалярного множення для багатоядерних процесорів.
6. Дослідіть та реалізуйте захист від side-channel атак для операцій ECC.
7. Створіть бібліотеку для роботи з кривими Едвардса та порівняйте з кривими Вейєрштрасса.
8. Реалізуйте threshold ECDSA для розподіленого підписування.
9. Дослідіть вплив різних координатних систем на продуктивність операцій.
10. Створіть систему для аудиту безпеки параметрів еліптичних кривих.

Теми лабораторних робіт

a. Реалізація арифметики еліптичних кривих та базових криптографічних примітивів.
b. Розробка PKI системи на основі ECDSA для управління сертифікатами.
c. Створення захищеного месенджера з використанням ECDH для обміну ключами.
d. Реалізація Bitcoin-подібної системи транзакцій з ECDSA підписами.
e. Розробка HSM емулятора з підтримкою операцій над еліптичними кривими.
f. Порівняльний аналіз різних стандартних кривих за безпекою та продуктивністю.
g. Реалізація пост-квантум безпечного протоколу на основі ізогеній еліптичних кривих.
h. Створення системи порогових підписів для мультипідписних гаманців.

i. Реалізація доведення з нульовим розголошенням на основі еліптичних кривих.

Криптографічні хеш-функції SHA-2 та SHA-3

Теоретичні відомості

Криптографічні хеш-функції є фундаментальними примітивами сучасної криптографії. Вони перетворюють повідомлення довільної довжини у значення фіксованої довжини, яке називається хеш-значенням або дайджестом. SHA (Secure Hash Algorithm) - це сімейство криптографічних хеш-функцій, розроблених Національним інститутом стандартів і технологій США (NIST).

Криптографічна хеш-функція H повинна задовольняти такі властивості:

1. Односпрямованість: Для заданого хеш-значення h обчислювально неможливо знайти повідомлення m таке, що H(m) = h.

2. Стійкість до колізій другого роду: Для заданого повідомлення m_1 обчислювально неможливо знайти інше повідомлення $m_2 \neq m_1$ таке, що $H(m_1) = H(m_2)$.

3. Стійкість до колізій: Обчислювально неможливо знайти будь-яку пару різних повідомлень m_1 та m_2 таких, що $H(m_1) = H(m_2)$.

4. Лавинний ефект: Навіть незначна зміна вхідного повідомлення призводить до повної зміни хеш-значення.

SHA-2: Архітектура та принципи роботи

SHA-2 - це сімейство хеш-функцій, що включає SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 та SHA-512/256. Число в назві вказує на довжину вихідного хеш-значення в бітах. Алгоритми SHA-256 та SHA-512 є основними, інші є їх модифікаціями.

Структура SHA-256

SHA-256 обробляє повідомлення блоками по 512 біт і виробляє 256-бітний хеш. Алгоритм складається з наступних етапів:

Етап 1: Підготовка повідомлення

Вхідне повідомлення доповнюється так, щоб його довжина була кратною 512 бітам:

```
Функція підготовкаПовідомлення(M):
    // M - вхідне повідомлення
    L = довжина(M) в бітах

    // Додаємо біт '1'
    M' = M || 1

    // Додаємо нулі до довжини L + 1 + k ≡ 448 (mod 512)
    k = (448 - (L + 1)) mod 512
    якщо k < 0:
        k = k + 512

    M'' = M' || 0^k  // додаємо k нулів

    // Додаємо 64-бітне представлення L
    M_padded = M'' || toBinary64(L)

    повернути M_padded
```

Етап 2: Ініціалізація хеш-значень

SHA-256 використовує вісім 32-бітних слів як початкові хеш-значення. Це перші 32 біти дробових частин квадратних коренів перших восьми простих чисел:

```
H₀ = 0x6a09e667  // √2
H₁ = 0xbb67ae85  // √3
H₂ = 0x3c6ef372  // √5
H₃ = 0xa54ff53a  // √7
H₄ = 0x510e527f  // √11
H₅ = 0x9b05688c  // √13
H₆ = 0x1f83d9ab  // √17
H₇ = 0x5be0cd19  // √19
```

Етап 3: Обробка блоків повідомлення

```
Алгоритм SHA256(повідомлення):
    // Підготовка
    M = підготовкаПовідомлення(повідомлення)
    N = довжина(M) / 512  // кількість блоків

    // Ініціалізація
    H = [H₀, H₁, H₂, H₃, H₄, H₅, H₆, H₇]

    // Константи (перші 32 біти дробових частин кубічних коренів перш
их 64 простих)
    K = ініціалізуватиКонстанти()
```

```
    // Обробка кожного блоку
    для i від 0 до N-1:
        блок = M[i*512 : (i+1)*512]
        H = обробитиБлок(блок, H, K)

    // Формування результату
    хеш = H[0] || H[1] || ... || H[7]
    повернути хеш
```

Етап 4: Обробка одного блоку

```
Функція обробитиБлок(блок, H, K):
    // Розширення повідомлення
    W = []

    // Перші 16 слів - це слова з блоку
    для t від 0 до 15:
        W[t] = блок[t*32 : (t+1)*32]

    // Розширення до 64 слів
    для t від 16 до 63:
        σ₀ = ROTR(W[t-15], 7) ⊕ ROTR(W[t-15], 18) ⊕ SHR(W[t-15], 3)
        σ₁ = ROTR(W[t-2], 17) ⊕ ROTR(W[t-2], 19) ⊕ SHR(W[t-2], 10)
        W[t] = W[t-16] + σ₀ + W[t-7] + σ₁ mod 2³²

    // Ініціалізація робочих змінних
    a, b, c, d, e, f, g, h = H[0], H[1], H[2], H[3], H[4], H[5], H[6]
, H[7]

    // Основний цикл
    для t від 0 до 63:
        Σ₁ = ROTR(e, 6) ⊕ ROTR(e, 11) ⊕ ROTR(e, 25)
        Ch = (e ∧ f) ⊕ (¬e ∧ g)
        T₁ = h + Σ₁ + Ch + K[t] + W[t]

        Σ₀ = ROTR(a, 2) ⊕ ROTR(a, 13) ⊕ ROTR(a, 22)
        Maj = (a ∧ b) ⊕ (a ∧ c) ⊕ (b ∧ c)
        T₂ = Σ₀ + Maj

        h = g
        g = f
        f = e
        e = d + T₁ mod 2³²
        d = c
        c = b
        b = a
```

```
        a = T₁ + T₂ mod 2³²

    // Додавання до попередніх значень
    H[0] = H[0] + a mod 2³²
    H[1] = H[1] + b mod 2³²
    H[2] = H[2] + c mod 2³²
    H[3] = H[3] + d mod 2³²
    H[4] = H[4] + e mod 2³²
    H[5] = H[5] + f mod 2³²
    H[6] = H[6] + g mod 2³²
    H[7] = H[7] + h mod 2³²

    повернути H
```

Допоміжні функції:

```
Функція ROTR(x, n):
    // Циклічний зсув вправо на n позицій
    повернути (x >> n) | (x << (32 - n))

Функція SHR(x, n):
    // Логічний зсув вправо на n позицій
    повернути x >> n

Функція Ch(x, y, z):
    // Choose function
    повернути (x ∧ y) ⊕ (¬x ∧ z)

Функція Maj(x, y, z):
    // Majority function
    повернути (x ∧ y) ⊕ (x ∧ z) ⊕ (y ∧ z)
```

SHA-3: Принципово інша архітектура

SHA-3 базується на конструкції губки та використовує функцію Keccak. На відміну від SHA-2, який використовує конструкцію Меркла-Дамгарда, SHA-3 має принципово іншу структуру.

Архітектура «губка»

1. Ідея губки

Keccak реалізовано так:

1. Absorb (всмоктування): комбінуємо (XOR) блоки вхідних даних із частиною внутрішнього стану.
2. Permute (перетворення): застосовуємо внутрішню багатоступінчасту перестановку f.

3. Після всіх блоків Squeeze (видавлювання): читаємо потрібну довжину виходу з частини стану, з можливістю продовжити «видавлювання».

2. Структура стану

- Стан має ширину b біт, розділену на дві частини:

- rate r — швидка частина (в смокчучи фазі XOR'иться з даними і з якої читається вихід)

- capacity c — здатність забезпечити стійкість до колізій/другого знайдення

- $b = r + c$. Наприклад, для SHA3-256 $b = 1600$, $r = 1088$, $c = 512$.

3. Перестановка Keccak-f

Ключова операція — Keccak-f[b], де $b \in \{25,50,100,200,400,800,1600\}$.

Стандарт використовує $b = 1600$.

Перестановка складається з $12+2\cdot\log_2(b/25)$ раундів (для $b = 1600$ — 24 раунди).

Кожен раунд виконує п'ять кроків:

1. **θ (Theta)** – змішує біти по стовпцях для рознесення залежностей.

2. **ρ (Rho)** – циклічні зсуви кожної «ланки» ($16 \times 16 \times 64$-бітова сітка) на фіксовані відстані.

3. **π (Pi)** – перестановка ланок у площині 5×5.

4. **χ (Chi)** – нелінійна операція, що змішує біти всередині кожного рядка.

5. **ι (Iota)** – XOR зі «слотною» константою, яка залежить від номера раунду.

Ці кроки гарантують високу дифузію й нелінійність.

Псевдокод такий:

```
Функція губка(повідомлення, довжинаВиходу, r, c):
    // r - швидкість (rate)
    // c - ємність (capacity)
    // b = r + c - ширина стану (для SHA-3: b = 1600)

    // Ініціалізація
    стан = [0] * b

    // Фаза поглинання
    блоки = розбитиНаБлоки(повідомлення || padding, r)
    для кожного блоку P:
```

```
        стан[0:r] = стан[0:r] ⊕ P
        стан = f(стан)  // функція перестановки

    // Фаза вичавлювання
    вихід = []
    поки довжина(вихід) < довжинаВиходу:
        вихід = вихід || стан[0:r]
        стан = f(стан)

    повернути вихід[0:довжинаВиходу]
```

Функція Keccak-f[1600]

Серце SHA-3 - це функція перестановки Keccak-f[1600], яка працює зі станом 1600 біт, організованим як тривимірний масив 5×5×64:

```
Функція Keccak_f(стан):
    // стан - масив 5×5×64 біт

    для раунд від 0 до 23:
        // Крок θ (Theta)
        стан = theta(стан)

        // Крок ρ (Rho)
        стан = rho(стан)

        // Крок π (Pi)
        стан = pi(стан)

        // Крок χ (Chi)
        стан = chi(стан)

        // Крок ι (Iota)
        стан = iota(стан, раунд)

    повернути стан
```

Детальний опис кроків:

```
Функція theta(A):
    // A[x,y,z] - біт у позиції (x,y,z)
    C = [][]  // масив 5×64
    D = [][]  // масив 5×64

    для x від 0 до 4:
        для z від 0 до 63:
            C[x][z] = A[x,0,z] ⊕ A[x,1,z] ⊕ A[x,2,z] ⊕ A[x,3,z] ⊕
A[x,4,z]
```

```
    для x від 0 до 4:
        для z від 0 до 63:
            D[x][z] = C[(x-1) mod 5][z] ⊕ C[(x+1) mod 5][(z-1) mod 6
4]

    для x від 0 до 4:
        для y від 0 до 4:
            для z від 0 до 63:
                A[x,y,z] = A[x,y,z] ⊕ D[x][z]

    повернути A

Функція rho(A):
    // Циклічні зсуви
    зсуви = [[0,1,62,28,27], [36,44,6,55,20], [3,10,43,25,39],
             [41,45,15,21,8], [18,2,61,56,14]]

    для x від 0 до 4:
        для y від 0 до 4:
            для z від 0 до 63:
                A'[x,y,z] = A[x,y,(z - зсуви[x][y]) mod 64]

    повернути A'

Функція pi(A):
    // Перестановка позицій
    для x від 0 до 4:
        для y від 0 до 4:
            для z від 0 до 63:
                A'[x,y,z] = A[(x + 3y) mod 5, x, z]

    повернути A'

Функція chi(A):
    // Нелінійне перетворення
    для x від 0 до 4:
        для y від 0 до 4:
            для z від 0 до 63:
                A'[x,y,z] = A[x,y,z] ⊕ ((¬A[(x+1) mod 5,y,z]) ∧ A[(x
+2) mod 5,y,z])

    повернути A'

Функція iota(A, раунд):
```

```
    // Додавання раундової константи
    RC = обчислитиРаундовуКонстанту(раунд)
    A[0,0] = A[0,0] ⊕ RC

    повернути A
```

Розглянемо обчислення **SHA-256 для повідомлення “abc”** (3 байти = 24 біти).

Крок 1: Підготовка повідомлення

Повідомлення “abc” в бітах:

01100001 01100010 01100011

Додаємо біт ‘1’:

01100001 01100010 01100011 1

Додаємо нулі до довжини 448 біт:

01100001 01100010 01100011 10000000 00000000 … (всього 424 нулі)

Додаємо довжину (24) як 64-бітне число:

… 00000000 00011000

Результат: один блок 512 біт.

Крок 2: Розширення повідомлення

W[0] = 0x61626380 (перші 4 байти з padding)

W[1] = 0x00000000

…

W[15] = 0x00000018 (довжина)

Для W[16] до W[63] застосовуємо формули розширення:

$W[16] = \sigma_1(W[14]) + W[9] + \sigma_0(W[1]) + W[0]$ …

Крок 3: Основний цикл

Початкові значення:

a = 0x6a09e667,

b = 0xbb67ae85,

c = 0x3c6ef372,

d = 0xa54ff53a

e = 0x510e527f,

f = 0x9b05688c,

g = 0x1f83d9ab,

h = 0x5be0cd19

Раунд 0:

- Σ_1 = ROTR(e, 6) ⊕ ROTR(e, 11) ⊕ ROTR(e, 25) = 0x2f7af88f
- Ch = (e ∧ f) ⊕ (¬e ∧ g) = 0x1b05200c
- T_1 = h + Σ_1 + Ch + K[0] + W[0] = 0xe07b3e7d
- Σ_0 = ROTR(a, 2) ⊕ ROTR(a, 13) ⊕ ROTR(a, 22) = 0x1ae21119
- Maj = (a ∧ b) ⊕ (a ∧ c) ⊕ (b ∧ c) = 0x3a67a665
- T_2 = Σ_0 + Maj = 0x5489b77e

Оновлення змінних:

h = g = 0x1f83d9ab

g = f = 0x9b05688c

…

a = T_1 + T_2 = 0x350575fb

Після 64 раундів та додавання початкових значень:

Результат

SHA-256(“abc”) =

ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad

Приклад роботи SHA-3-256

Для SHA-3-256 використовується r = 1088 біт, c = 512 біт.

Крок 1: Підготовка

Повідомлення “abc” доповнюється специфічним для SHA-3 padding:

“abc” || 0x06 || 0x00…00 || 0x80

Крок 2: Поглинання

Ініціалізуємо стан нулями та XOR з першим (і єдиним) блоком.

Крок 3: Застосування Keccak-f[1600]

24 раунди перетворень θ, ρ, π, χ, ι.

Крок 4: Вичавлювання

Беремо перші 256 біт результату.

Результат

SHA-3-256(“abc”) =

3a985da74fe225b2045c172d6bd390bd855f086e3e9d525b46bfe24511431532

Область застосування

SHA-2 та SHA-3 широко використовуються в різних областях криптографії та інформаційної безпеки:

- Цифрові підписи: Використовуються в алгоритмах DSA, ECDSA, RSA-PSS для створення дайджестів повідомлень перед підписанням.

- Сертифікати та PKI: Всі сучасні SSL/TLS сертифікати використовують SHA-256 або вище для підписання.

- Blockchain та криптовалюти: Bitcoin використовує подвійне хешування SHA-256 для proof-of-work. Ethereum перейшов на Keccak-256 (попередник SHA-3).

- Зберігання паролів: Хоча для цього краще підходять спеціалізовані функції як bcrypt або Argon2, SHA-2/3 часто використовуються як компоненти в схемах хешування паролів.

- Перевірка цілісності: Контрольні суми файлів, верифікація завантажень, системи виявлення вторгнень.

- Генерація випадкових чисел: Використовуються в PRNG та системах виведення ключів (KDF).

- HMAC: Hash-based Message Authentication Code використовує SHA-2/3 для автентифікації повідомлень.

Приклади застосування

Приклад 1: Перевірка цілісності файлів

Система автоматичного оновлення програмного забезпечення використовує SHA-256 для перевірки цілісності завантажених файлів. Сервер публікує хеш-значення кожного файлу, а клієнт після завантаження обчислює хеш і порівнює з еталонним. Це захищає від підміни файлів під час передачі або на дзеркалах завантаження.

Приклад 2: Blockchain Mining

У Bitcoin кожен блок містить SHA-256 хеш попереднього блоку, створюючи ланцюг. Майнери шукають nonce таке, щоб SHA-256(SHA-256(block_header)) був менше цільового значення. Подвійне хешування захищає від атак розширення довжини.

Приклад 3: Password-Based Key Derivation

Системи шифрування дисків використовують PBKDF2 з SHA-512 для перетворення паролів користувачів у криптографічні ключі. Багаторазове застосування хеш-функції ускладнює атаки перебору.

Переваги та недоліки

Переваги SHA-2

- Широка підтримка: Реалізований у всіх основних криптографічних бібліотеках та апаратних прискорювачах.

- Перевірена безпека: Більше 20 років криптоаналізу без знайдених практичних атак.

- Гнучкість: Різні варіанти (SHA-224/256/384/512) для різних рівнів безпеки.

- Швидкість: Ефективні реалізації, особливо на 32/64-бітних процесорах.

- Стандартизація: FIPS стандарт, підтримується всіма регуляторами.

Недоліки SHA-2

- Вразливість до атак розширення довжини: Властивість конструкції Меркла-Дамгарда.

- Складність реалізації: Багато раундів та операцій, що збільшує ймовірність помилок.

- Неоптимальність для коротких повідомлень: Фіксований оверхед padding.

- Потенційна вразливість: Схожа структура з SHA-1, який вже зламаний.

Переваги SHA-3

- Інноваційний дизайн: Конструкція губки надає додаткові можливості (XOF, паралельність).

- Стійкість до атак розширення: Конструкція губки не має цієї вразливості.

- Гнучкість виходу: SHAKE варіанти дозволяють довільну довжину виходу.

- Простота аналізу: Чіткі математичні основи полегшують криптоаналіз.

- Резерв на майбутнє: Альтернатива SHA-2 у разі знайдення вразливостей.

Недоліки SHA-3

- Повільніша швидкість: На деяких платформах повільніша за SHA-2.

- Менша поширеність: Не всі системи мають native підтримку.

- Більше використання пам’яті: Стан 1600 біт проти 256/512 для SHA-2.

- Складність оптимізації: Менше можливостей для апаратного прискорення.

Контрольні запитання

- Які основні відмінності між конструкцією Меркла-Дамгарда (SHA-2) та конструкцією губки (SHA-3)?
- Чому SHA-256 використовує константи, похідні від квадратних та кубічних коренів простих чисел?
- Яка роль функцій Ch та Maj в алгоритмі SHA-256?
- Поясніть призначення кожного з п'яти кроків (θ, ρ, π, χ, ι) у функції Keccak-f.
- Чому важливо правильно реалізувати padding в хеш-функціях?
- Як параметри r (rate) та c (capacity) впливають на безпеку SHA-3?
- Які переваги дає використання SHAKE функцій порівняно з фіксованим виходом?
- Чому в Bitcoin використовується подвійне хешування SHA-256?
- Як атака розширення довжини працює на SHA-2 і чому вона неможлива на SHA-3?
- Які критерії використовувалися при виборі SHA-3 на конкурсі NIST?

Методичні вказівки до лабораторної роботи

Назва

Реалізація та дослідження криптографічних хеш-функцій SHA-2 та SHA-3

Мета

Вивчити принципи роботи сучасних криптографічних хеш-функцій, реалізувати алгоритми SHA-256 та SHA-3-256, дослідити їх властивості та порівняти продуктивність.

Завдання

- Реалізувати повний алгоритм SHA-256 включно з padding та обробкою блоків

- Реалізувати функцію Keccak-f[1600] та на її основі SHA-3-256

- Створити модуль для перевірки основних криптографічних властивостей

- Реалізувати HMAC на основі обох хеш-функцій

- Провести аналіз продуктивності для різних розмірів повідомлень

- Продемонструвати лавинний ефект

- Реалізувати пошук часткових колізій (перші n біт)

Опис

Лабораторна робота передбачає покрокову реалізацію двох принципово різних підходів до побудови хеш-функцій. Студенти повинні реалізувати всі компоненти алгоритмів, включаючи допоміжні функції, та продемонструвати розуміння їх призначення.

Порядок виконання

- Створити модуль бітових операцій (зсуви, XOR, AND, NOT)
- Реалізувати функцію padding для SHA-256 та SHA-3
- Реалізувати допоміжні функції SHA-256 (Ch, Maj, Σ_0, Σ_1, σ_0, σ_1)
- Реалізувати розширення повідомлення та основний цикл SHA-256
- Реалізувати п’ять кроків функції Keccak-f (θ, ρ, π, χ, ι)
- Реалізувати конструкцію губки для SHA-3
- Створити тестовий набір з відомими векторами
- Реалізувати візуалізацію внутрішнього стану
- Провести benchmarking обох алгоритмів
- Реалізувати демонстрацію властивостей

Результат

Працююча реалізація обох хеш-функцій з можливістю обробки файлів довільного розміру, візуалізацією процесу хешування та аналізом криптографічних властивостей.

Після виконання лабораторної роботи студент повинен надати:

- Вихідний код програми, де чітко виділені:
- Модуль бітових операцій
- Функції padding для SHA-2 та SHA-3
- Реалізація SHA-256 з усіма проміжними функціями
- Реалізація Keccak-f[1600] з п’ятьма кроками
- Реалізація конструкції губки SHA-3
- Модуль тестування з еталонними векторами
- Модуль аналізу властивостей
- Продемонструвати роботу додатку:
- Правильне хешування тестових векторів NIST

- Обробку файлів різного розміру
- Візуалізацію лавинного ефекту
- Порівняння швидкості SHA-256 та SHA-3-256
- Пошук часткових колізій для малої кількості біт
- Роботу HMAC-SHA256 та HMAC-SHA3-256

Бути готовим відповісти на питання:

- Як працює розширення повідомлення в SHA-256?
- Чому використовується саме 64/80 раундів?
- Як забезпечується дифузія в функції Keccak?
- Які атаки можливі при неправильній реалізації?
- Як вибрати між SHA-2 та SHA-3 для конкретного застосування?

Тести

- Яка довжина внутрішнього стану SHA-3?
- 1) 256 біт
- 2) 512 біт
- 3) 1024 біти
- 4) 1600 біт
- Скільки раундів використовує SHA-256?
- 1) 48
- 2) 64
- 3) 80
- 4) 24
- Яка конструкція лежить в основі SHA-3?
- 1) Меркла-Дамгарда
- 2) Девіса-Мейєра
- 3) Губки
- 4) Фейстеля
- Що означає число в назві SHA-512?
- 1) Розмір блоку
- 2) Кількість раундів

- 3) Довжину виходу в бітах
- 4) Розмір внутрішнього стану
- Яка функція НЕ використовується в SHA-256?
- 1) Ch (Choose)
- 2) Maj (Majority)
- 3) θ (Theta)
- 4) Σ (Sigma)

Відповіді:

1-4,

2-2,

3-3,

4-3,

5-3

Тести підвищеної складності

- Який розмір блоку використовує SHA-512?
- 1) 512 біт
- 2) 1024 біти
- 3) 2048 біт
- 4) 256 біт
- Яке значення capacity (c) використовується в SHA3-256?
- 1) 256 біт
- 2) 512 біт
- 3) 1024 біти
- 4) 1088 біт
- Скільки раундів у функції Keccak-f[1600]?
- 1) 12
- 2) 16
- 3) 20
- 4) 24
- Яка атака можлива на SHA-2, але не на SHA-3?

- 1) Атака днів народження
- 2) Атака розширення довжини
- 3) Диференційна атака
- 4) Квантова атака
- Що таке SHAKE256?
- 1) Варіант SHA-256 з іншими константами
- 2) SHA-3 з розширюваним виходом
- 3) Комбінація SHA-2 та SHA-3
- 4) Хеш-функція для квантової стійкості
- Яка складність найкращої відомої атаки на пошук прообразу SHA-256?
- 1) 2^128
- 2) 2^256
- 3) 2^512
- 4) 2^64
- Чому в SHA-256 використовується доповнення до довжини кратної 512?
- 1) Для оптимізації швидкості
- 2) Це розмір блоку обробки
- 3) Для сумісності з SHA-1
- 4) Випадковий вибір
- Який крок Кессак відповідає за нелінійність?
- 1) θ (theta)
- 2) ρ (rho)
- 3) π (pi)
- 4) χ (chi)
- Скільки 32-бітних слів у внутрішньому стані SHA-256?
- 1) 5
- 2) 8
- 3) 16
- 4) 64
- Що означає “3” в SHA-3?

- 1) Третя версія SHA

- 2) Третє покоління хеш-функцій

- 3) Переможець третього раунду конкурсу

- 4) Результат конкурсу на SHA третього покоління

Відповіді:

1-2. SHA-512 обробляє блоки по 1024 біти, в два рази більше ніж SHA-256

2-2. SHA3-256 використовує capacity 512 біт, rate = 1600 - 512 = 1088 біт

3-4. Keccak-f[1600] виконує 24 раунди перетворень

4-2. Атака розширення довжини можлива через конструкцію Меркла-Дамгарда

5-2. SHAKE256 - це SHA-3 варіант з довільною довжиною виходу (XOF)

6-2. Найкраща атака - повний перебір з складністю 2^256

7-2. 512 біт - це розмір блоку, який обробляє SHA-256 за одну ітерацію

8-4. Крок χ (chi) - єдиний нелінійний крок в Keccak

9-2. SHA-256 має 8 внутрішніх 32-бітних регістрів (h0-h7)

10-4. SHA-3 - результат відкритого конкурсу NIST на нове покоління хеш-функцій

Завдання для самостійної роботи

1. Реалізуйте SHA-224 та SHA-384 на основі вашої реалізації SHA-256 та SHA-512, визначивши необхідні модифікації.
2. Дослідіть та реалізуйте оптимізації SHA-256 з використанням SIMD інструкцій процесора.
3. Створіть програму для пошуку часткових колізій методом днів народження та оцініть практичну складність.
4. Реалізуйте SHAKE128 та SHAKE256 - варіанти SHA-3 з довільною довжиною виходу.
5. Розробіть візуалізацію поширення змін через раунди SHA-256 та Keccak для демонстрації лавинного ефекту.
6. Реалізуйте паралельну версію SHA-3 для обробки декількох повідомлень одночасно.
7. Створіть benchmark suite для порівняння різних реалізацій на різних платформах.
8. Дослідіть вплив різних методів padding на безпеку та продуктивність.

9. Реалізуйте спрощені версії SHA-2 та SHA-3 зі зменшеною кількістю раундів та проаналізуйте їх стійкість.
10. Створіть криптографічний PRNG на основі SHA-3 в режимі губки.

Теми лабораторних робіт

Розробка системи перевірки цілісності файлів з використанням SHA-2/3 та Merkle trees.

Реалізація proof-of-work системи на базі SHA-256 з адаптивною складністю.

Створення менеджера паролів з використанням PBKDF2-SHA512 для деривації ключів.

Розробка системи цифрових підписів з використанням SHA-3 та еліптичних кривих.

Реалізація розподіленої хеш-таблиці з криптографічними гарантіями на SHA-256.

Створення blockchain прототипу з вибором між SHA-2 та SHA-3 для різних компонентів.

Розробка системи дедуплікації даних з використанням convergent encryption та SHA-256.

Реалізація протоколу обміну ключами з використанням SHA-3 як KDF.

Створення системи моніторингу змін конфігурації з криптографічними контрольними сумами.

Розробка бібліотеки для постквантових хеш-функцій на основі SHA-3.

Криптографічна хеш-функція BLAKE2

Теоретичні відомості

BLAKE2 є сучасною криптографічною хеш-функцією, яка була розроблена як покращена версія BLAKE - фіналіста конкурсу SHA-3. Алгоритм був представлений у 2012 році Жан-Філіпом Омассоном, Самюелем Невесом, Крістіаном Вінненом та Зоко Вілкокс-О'Хірном. BLAKE2 оптимізований для високої швидкості на сучасних процесорах, забезпечуючи при цьому високий рівень безпеки, порівнянний з SHA-3.

BLAKE2 існує у двох основних варіантах:

- **BLAKE2b**: оптимізований для 64-бітних платформ, виробляє хеш до 512 біт

- **BLAKE2s**: оптимізований для 32-бітних платформ, виробляє хеш до 256 біт

Алгоритм базується на конструкції HAIFA (HAsh Iterative FrAmework) та використовує модифіковану версію потокового шифру ChaCha. Основна інновація BLAKE2 полягає в тому, що він досягає швидкості, порівнянної з нескриптографічними хеш-функціями, зберігаючи при цьому криптографічну стійкість.

Розглянемо покроковий опис основного алгоритму BLAKE2b

Крок 1: Початкова підготовка та прийом параметрів

Основний алгоритм BLAKE2b починається з прийняття трьох параметрів: самого повідомлення, яке потрібно захешувати, опціонального ключа для режиму автентифікації та бажаної довжини вихідного хеш-значення. Якщо довжина виходу не вказана явно, алгоритм використовує значення за замовчуванням - 64 байти, що відповідає максимальній довжині для BLAKE2b.

Крок 2: Ініціалізація структури параметрів

Першим кроком алгоритму є ініціалізація структури параметрів. Ця структура містить всі налаштування, необхідні для роботи хеш-функції. Алгоритм встановлює довжину дайджесту відповідно до запитаної довжини виходу, записує довжину ключа (якщо ключ присутній), встановлює параметри fanout та depth в значення 1 (що означає послідовну обробку без дерева хешування), обнуляє параметри листа дерева та зміщення вузла, а також заповнює нулями масиви солі та персоналізації розміром по 16 байтів кожен.

Крок 3: Ініціалізація внутрішнього стану

Після підготовки параметрів алгоритм переходить до ініціалізації внутрішнього стану. Стан представлений масивом з восьми 64-бітних слів, які спочатку копіюються з констант ініціалізаційного вектора IV. Ці константи являють собою дробові частини квадратних коренів перших восьми простих чисел. Потім перше слово стану модифікується шляхом виконання операції XOR з упакованим 64-бітним представленням структури параметрів. Це забезпечує, що різні конфігурації параметрів дають різні хеш-функції.

Крок 4: Обробка ключа

Якщо користувач надав ключ для режиму MAC, алгоритм виконує спеціальну обробку. Створюється тимчасовий блок розміром 128 байтів, який повністю заповнюється нулями. Ключ копіюється на початок цього

блоку, а решта байтів залишаються нульовими. Цей блок обробляється функцією стиснення F як перший блок даних, при цьому лічильник встановлюється в 128 (що вказує на обробку 128 байтів), а прапори фіналізації встановлені в 0. Результат цієї операції оновлює внутрішній стан h.

Крок 5: Підготовка до обробки повідомлення

Далі алгоритм переходить до основної фази обробки повідомлення. Повідомлення обробляється блоками по 128 байтів. Алгоритм використовує змінну для відстеження залишку необроблених даних та лічильник для підрахунку загальної кількості оброблених байтів.

Крок 6: Основний цикл обробки повних блоків

Основний цикл обробки працює доти, доки в залишку є більше одного повного блоку (128 байтів). На кожній ітерації алгоритм витягує перші 128 байтів із залишку, формує з них блок для обробки та видаляє ці байти з залишку. Лічильник збільшується на 128, відображаючи кількість оброблених байтів. Блок передається функції стиснення F разом з поточним станом, значенням лічильника та нульовими прапорами фіналізації. Функція стиснення оновлює внутрішній стан h.

Крок 7: Підготовка останнього блоку

Після обробки всіх повних блоків залишається останній неповний блок (або порожній, якщо повідомлення було кратне 128 байтам). Для обробки цього останнього блоку алгоритм створює новий блок розміром 128 байтів, заповнений нулями. Залишкові дані копіюються на початок цього блоку, а решта позицій залишаються нульовими. Це забезпечує, що функція стиснення завжди отримує повний блок даних.

Крок 8: Фіналізація - обробка останнього блоку

Лічильник оновлюється, додаючи фактичну кількість байтів у залишку (не 128, а реальний розмір останніх даних). Це критично важливо для правильної роботи алгоритму, оскільки лічильник відображає точну довжину оригінального повідомлення.

Останній блок обробляється функцією стиснення з особливим прапором фіналізації, встановленим у значення 0xFFFFFFFFFFFFFFFF (всі біти встановлені в 1). Цей прапор сигналізує функції стиснення, що це останній блок повідомлення, що може вплинути на внутрішню обробку для забезпечення правильної фіналізації.

Крок 9: Формування вихідного масиву

Після обробки всіх блоків алгоритм переходить до формування вихідного значення. Внутрішній стан h містить вісім 64-бітних слів, що в сумі дає 512 біт даних. Алгоритм створює вихідний масив та копіює необхідну кількість слів зі стану. Кожне слово конвертується в little-endian формат (молодший байт першим) перед додаванням до вихідного масиву.

Крок 10: Обрізання до запитаної довжини

Кількість слів для копіювання визначається запитаною довжиною виходу, поділеною на 8 (оскільки кожне слово містить 8 байтів). Наприклад, для стандартного 512-бітного (64-байтного) виходу копіюються всі 8 слів. Для 256-бітного (32-байтного) виходу копіюються тільки перші 4 слова.

Нарешті, алгоритм обрізає вихідний масив до точної запитаної довжини. Це необхідно для випадків, коли довжина виходу не кратна 8 байтам. Наприклад, якщо користувач запитав 30-байтний хеш, алгоритм скопіює перші 4 слова (32 байти) і потім обріже результат до 30 байтів.

Важливі особливості реалізації

Важливо відзначити, що алгоритм BLAKE2b не використовує традиційне доповнення, як це роблять SHA-2 або MD5. Замість додавання спеціальних бітів та довжини повідомлення в кінці, BLAKE2b покладається на лічильник байтів, який передається функції стиснення. Це спрощує реалізацію та усуває деякі види атак, пов’язаних з доповненням.

Підтримка інкрементального хешування

Також варто підкреслити, що алгоритм підтримує інкрементальне хешування - можна обробляти повідомлення частинами, зберігаючи стан між викликами. Це особливо корисно для потокової обробки великих файлів або даних, що надходять по мережі.

Оптимізація для сучасних процесорів

Весь процес спроектований для максимальної ефективності на сучасних 64-бітних процесорах, використовуючи природний розмір машинного слова та уникаючи операцій, які можуть сповільнити виконання. Результатом є криптографічно стійка хеш-функція, яка працює швидше за багато інших популярних алгоритмів.

Архітектура BLAKE2b

BLAKE2b працює з блоками по 128 байтів (1024 біти) та має внутрішній стан з 8 слів по 64 біти. Розглянемо детальну структуру алгоритму.

Параметри та константи

```
Константи BLAKE2b:
    // Початкові вектори (дробові частини квадратних коренів простих чисел)
    IV[0] = 0x6a09e667f3bcc908  // √2
    IV[1] = 0xbb67ae8584caa73b  // √3
    IV[2] = 0x3c6ef372fe94f82b  // √5
    IV[3] = 0xa54ff53a5f1d36f1  // √7
    IV[4] = 0x510e527fade682d1  // √11
    IV[5] = 0x9b05688c2b3e6c1f  // √13
    IV[6] = 0x1f83d9abfb41bd6b  // √17
    IV[7] = 0x5be0cd19137e2179  // √19

    // Перестановки для функції G
    SIGMA = [
        [0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14, 15],
        [14, 10, 4,  8,  9,  15, 13, 6,  1,  12, 0,  2,  11, 7,  5,  3],
        [11, 8,  12, 0,  5,  2,  15, 13, 10, 14, 3,  6,  7,  1,  9,  4],
        [7,  9,  3,  1,  13, 12, 11, 14, 2,  6,  5,  10, 4,  0,  15, 8],
        [9,  0,  5,  7,  2,  4,  10, 15, 14, 1,  11, 12, 6,  8,  3,  13],
        [2,  12, 6,  10, 0,  11, 8,  3,  4,  13, 7,  5,  15, 14, 1,  9],
        [12, 5,  1,  15, 14, 13, 4,  10, 0,  7,  6,  3,  9,  2,  8,  11],
        [13, 11, 7,  14, 12, 1,  3,  9,  5,  0,  15, 4,  8,  6,  2,  10],
        [6,  15, 14, 9,  11, 3,  0,  8,  12, 2,  13, 7,  1,  4,  10, 5],
        [10, 2,  8,  4,  7,  6,  1,  5,  15, 11, 9,  14, 3,  12, 13, 0],
        [0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10, 11, 12, 13, 14, 15],
        [14, 10, 4,  8,  9,  15, 13, 6,  1,  12, 0,  2,  11, 7,  5,  3]
    ]
```

Функція стиснення

Серцем BLAKE2 є функція стиснення F, яка приймає стан h, блок повідомлення m, лічильник t та прапори f:

```
Функція F(h, m, t, f):
    // h - стан (8 слів по 64 біти)
    // m - блок повідомлення (16 слів по 64 біти)
    // t - лічильник байтів (128 біт як два 64-бітних слова)
    // f - прапори фіналізації (два 64-бітних слова)

    // Ініціалізація робочого вектора
    v[0..7] = h[0..7]
    v[8] = IV[0]
    v[9] = IV[1]
    v[10] = IV[2]
    v[11] = IV[3]
    v[12] = IV[4] ⊕ t[0]  // молодші 64 біти лічильника
```

```
    v[13] = IV[5] ⊕ t[1]  // старші 64 біти лічильника
    v[14] = IV[6] ⊕ f[0]  // прапор фіналізації
    v[15] = IV[7] ⊕ f[1]  // зарезервовано

    // 12 раундів
    для r від 0 до 11:
        // Застосування функції G до стовпців
        G(v, 0, 4, 8,  12, m[SIGMA[r][0]], m[SIGMA[r][1]])
        G(v, 1, 5, 9,  13, m[SIGMA[r][2]], m[SIGMA[r][3]])
        G(v, 2, 6, 10, 14, m[SIGMA[r][4]], m[SIGMA[r][5]])
        G(v, 3, 7, 11, 15, m[SIGMA[r][6]], m[SIGMA[r][7]])

        // Застосування функції G до діагоналей
        G(v, 0, 5, 10, 15, m[SIGMA[r][8]],  m[SIGMA[r][9]])
        G(v, 1, 6, 11, 12, m[SIGMA[r][10]], m[SIGMA[r][11]])
        G(v, 2, 7, 8,  13, m[SIGMA[r][12]], m[SIGMA[r][13]])
        G(v, 3, 4, 9,  14, m[SIGMA[r][14]], m[SIGMA[r][15]])

    // Фіналізація
    для i від 0 до 7:
        h[i] = h[i] ⊕ v[i] ⊕ v[i+8]

    повернути h
```

Функція змішування G

Функція G виконує основні операції змішування:

```
Функція G(v, a, b, c, d, x, y):
    // Константи ротації
    R1 = 32
    R2 = 24
    R3 = 16
    R4 = 63

    v[a] = v[a] + v[b] + x
    v[d] = ROTR(v[d] ⊕ v[a], R1)

    v[c] = v[c] + v[d]
    v[b] = ROTR(v[b] ⊕ v[c], R2)

    v[a] = v[a] + v[b] + y
    v[d] = ROTR(v[d] ⊕ v[a], R3)

    v[c] = v[c] + v[d]
    v[b] = ROTR(v[b] ⊕ v[c], R4)
```

Псевдокод основного алгоритму BLAKE2b такий:

```
Алгоритм BLAKE2b(повідомлення, ключ, довжинаВиходу):
    // Параметри за замовчуванням
    якщо довжинаВиходу не задана:
        довжинаВиходу = 64  // байтів

    // Ініціалізація параметрів
    P = структураПараметрів()
    P.digestLength = довжинаВиходу
    P.keyLength = довжина(ключ)
    P.fanout = 1
    P.depth = 1
    P.leafLength = 0
    P.nodeOffset = 0
    P.nodeDepth = 0
    P.innerLength = 0
    P.salt = [0] * 16
    P.personal = [0] * 16

    // Ініціалізація стану
    h = копія(IV)
    h[0] = h[0] ⊕ упакувати64(P)

    // Обробка ключа (якщо є)
    якщо довжина(ключ) > 0:
        блок = [0] * 128
        копіювати(ключ, блок)
        h = F(h, блок, 128, 0)

    // Обробка повідомлення
    залишок = повідомлення
    лічильник = 0

    поки довжина(залишок) > 128:
        блок = залишок[0:128]
        залишок = залишок[128:]
        лічильник = лічильник + 128
        h = F(h, блок, лічильник, 0)

    // Обробка останнього блоку
    блок = [0] * 128
    копіювати(залишок, блок)
    лічильник = лічильник + довжина(залишок)
    h = F(h, блок, лічильник, 0xFFFFFFFFFFFFFFFF)  // прапор останньо
го блоку
```

```
    // Формування виходу
    вихід = []
    для і від 0 до довжинаВиходу/8 - 1:
        вихід.додати(toLittleEndian64(h[i]))

    повернути вихід[0:довжинаВиходу]
```

Розглянемо обчислення BLAKE2b для короткого повідомлення “abc” з виходом 64 байти.

Крок 1: Ініціалізація

Параметри: - Повідомлення: “abc” = [0x61, 0x62, 0x63] (3 байти) - Довжина виходу: 64 байти - Ключ: відсутній

Ініціалізація стану:

```
h[0] = IV[0] ⊕ 0x01010040 = 0x6a09e667f3bcc908 ⊕ 0x01010040 = 0x6b0
8e627f3bcc908
h[1] = IV[1] = 0xbb67ae8584caa73b
h[2] = IV[2] = 0x3c6ef372fe94f82b
h[3] = IV[3] = 0xa54ff53a5f1d36f1
h[4] = IV[4] = 0x510e527fade682d1
h[5] = IV[5] = 0x9b05688c2b3e6c1f
h[6] = IV[6] = 0x1f83d9abfb41bd6b
h[7] = IV[7] = 0x5be0cd19137e2179
```

Крок 2: Підготовка блоку

Оскільки повідомлення коротке (3 байти), воно поміщається в один блок:

```
m[0] = 0x0000000000636261  // "abc" в little-endian
m[1] = 0x0000000000000000
m[2] = 0x0000000000000000
...
m[15] = 0x0000000000000000
```

Крок 3: Виклик функції стиснення

Викликаємо F(h, m, 3, 0xFFFFFFFFFFFFFFFF) з: - t[0] = 3 (обробили 3 байти) - t[1] = 0 - f[0] = 0xFFFFFFFFFFFFFFFF (останній блок) - f[1] = 0

Крок 4: Виконання раундів

Ініціалізація робочого вектора v:

```
v[0..7] = h[0..7]
v[8] = IV[0] = 0x6a09e667f3bcc908
v[9] = IV[1] = 0xbb67ae8584caa73b
v[10] = IV[2] = 0x3c6ef372fe94f82b
v[11] = IV[3] = 0xa54ff53a5f1d36f1
v[12] = IV[4] ⊕ 3 = 0x510e527fade682d2
```

```
v[13] = IV[5] ⊕ 0 = 0x9b05688c2b3e6c1f
v[14] = IV[6] ⊕ 0xFFFFFFFFFFFFFFFF = 0xe07c2654049e4294
v[15] = IV[7] ⊕ 0 = 0x5be0cd19137e2179
```
Виконуємо 12 раундів з функціями G згідно з перестановками SIGMA.

Крок 5: Результат

Після всіх раундів та фіналізації отримуємо:

BLAKE2b("abc") = ba80a53f981c4d0d6a2797b69f12f6e94c212f14685ac4b74b12bb6fdbffa2d17d87c5392aab792dc252d5de4533cc9518d38aa8dbf1925ab92386edd4009923

Варіації та режими роботи

1. BLAKE2s

BLAKE2s - версія для 32-бітних платформ з наступними відмінностями: - Блоки по 64 байти замість 128 - 32-бітні слова замість 64-бітних - 10 раундів замість 12 - Інші константи ротації в функції G

2. BLAKE2bp та BLAKE2sp

Паралельні версії алгоритму, які можуть обробляти дані на декількох ядрах: - BLAKE2bp - паралельна версія BLAKE2b (4 паралельні екземпляри) - BLAKE2sp - паралельна версія BLAKE2s (8 паралельних екземплярів)

3. Режим з ключем (MAC)

BLAKE2 може працювати як MAC (Message Authentication Code):

```
Функція BLAKE2_MAC(повідомлення, ключ):
    якщо довжина(ключ) > 64:  // для BLAKE2b
        помилка("Ключ занадто довгий")

    повернути BLAKE2b(повідомлення, ключ, 64)
```

Область застосування

BLAKE2 широко використовується в сучасних системах завдяки своїй високій швидкості та безпеці:

- Криптовалюти та блокчейн: Siacoin, Nano та інші проекти використовують BLAKE2b для proof-of-work або хешування транзакцій. Zcash використовує BLAKE2 в протоколі Equihash.

- Системи контролю версій: Git експериментально підтримує BLAKE2b як альтернативу SHA-1. Mercurial планує перехід на BLAKE2b.

- Файлові системи: BLAKE2 використовується в ZFS для контрольних сум, в IPFS для адресації контенту, в Btrfs як опція для контрольних сум.

- Мережеві протоколи: WireGuard VPN використовує BLAKE2s для автентифікації пакетів. Noise Protocol Framework підтримує BLAKE2 для хешування.

- Менеджери пакетів: BLAKE2 використовується в різних системах управління пакетами для перевірки цілісності завантажень.

- Бази даних: Деякі NoSQL бази даних використовують BLAKE2 для внутрішнього хешування та індексації.

Приклади застосування

Приклад 1: Перевірка цілісності великих файлів

Система резервного копіювання використовує BLAKE2b для швидкої перевірки цілісності великих архівів. Завдяки високій швидкості (більше 1 ГБ/с на сучасних процесорах), BLAKE2b дозволяє перевіряти терабайтні архіви за розумний час. Система зберігає хеш кожного файлу та може швидко виявити пошкоджені або змінені дані.

Приклад 2: Автентифікація в IoT пристроях

Мережа IoT сенсорів використовує BLAKE2s для автентифікації повідомлень. Кожен сенсор має унікальний ключ та використовує BLAKE2s в режимі MAC для підписування даних. Малий розмір коду та низькі вимоги до пам’яті роблять BLAKE2s ідеальним для обмежених пристроїв.

Приклад 3: Дедуплікація даних в хмарному сховищі

Хмарна система зберігання використовує BLAKE2bp для ідентифікації дублікатів файлів. Паралельна версія дозволяє ефективно використовувати багатоядерні сервери для обробки великих обсягів даних. Система обчислює хеш кожного блоку даних та зберігає тільки унікальні блоки.

Переваги та недоліки

Переваги

- Висока швидкість: BLAKE2b швидший за MD5, SHA-1, SHA-2 та SHA-3 на сучасних процесорах. На 64-бітних платформах досягає швидкості понад 1 ГБ/с.

- Безпека: Базується на перевіреному дизайні BLAKE, який пройшов інтенсивний криптоаналіз під час конкурсу SHA-3. Немає відомих криптографічних вразливостей.

- Гнучкість: Підтримує довільну довжину виходу (1-64 байти для BLAKE2b, 1-32 для BLAKE2s). Вбудована підтримка ключів для MAC режиму.

- Простота реалізації: Простіший за SHA-2 та SHA-3, менше можливостей для помилок. Не потребує великих таблиць констант.

- Паралелізм: BLAKE2bp та BLAKE2sp дозволяють ефективно використовувати багатоядерні процесори.

- Малі вимоги до пам'яті: BLAKE2s потребує менше 1 КБ RAM, що важливо для вбудованих систем.

Недоліки

- Менша поширеність: Не всі платформи мають native підтримку BLAKE2. Менше апаратних прискорювачів порівняно з SHA-2.

- Відсутність стандартизації NIST: На відміну від SHA-2/3, BLAKE2 не є офіційним стандартом NIST, що може бути перешкодою для деяких застосувань.

- Складність паралельних версій: BLAKE2bp/sp складніші в реалізації та аналізі безпеки.

- Обмежений розмір ключа: Максимум 64 байти для BLAKE2b та 32 для BLAKE2s, що може бути недостатньо для деяких застосувань.

Контрольні запитання

1. Які основні відмінності між BLAKE2b та BLAKE2s?

2. Чому BLAKE2 швидший за SHA-2 та SHA-3?

3. Яка роль функції G в алгоритмі BLAKE2?

4. Як BLAKE2 може працювати в режимі MAC?

5. Що таке прапор фіналізації та навіщо він потрібен?

6. Які переваги дає використання конструкції HAIFA в BLAKE2?

7. Чому в BLAKE2 використовується 12 раундів замість 14 як у BLAKE?

8. Як працюють паралельні версії BLAKE2bp та BLAKE2sp?

9. Які константи ротації використовуються в функції G та чому саме вони?

10. Як BLAKE2 захищається від атак розширення довжини?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження криптографічної хеш-функції BLAKE2

2. Мета

Вивчити архітектуру сучасної високошвидкісної хеш-функції BLAKE2, реалізувати алгоритми BLAKE2b та BLAKE2s, дослідити їх продуктивність та криптографічні властивості.

3. Завдання

- Реалізувати функцію змішування G для BLAKE2b та BLAKE2s
- Реалізувати функцію стиснення F з правильною обробкою лічильників
- Реалізувати повні алгоритми BLAKE2b та BLAKE2s
- Додати підтримку режиму з ключем (MAC)
- Реалізувати паралельну версію BLAKE2bp або BLAKE2sp
- Провести тестування на векторах з RFC 7693
- Порівняти продуктивність з SHA-256 та SHA-3

4. Опис

Лабораторна робота передбачає покрокову реалізацію BLAKE2 від базових операцій до повного алгоритму. Особлива увага приділяється правильній реалізації little-endian арифметики та обробці граничних випадків.

5. Порядок виконання

- Реалізувати допоміжні функції (ротація, little-endian конвертація)
- Реалізувати функцію G з правильними константами ротації
- Реалізувати ініціалізацію параметрів та стану
- Реалізувати функцію стиснення F
- Реалізувати обробку повідомлень з правильним padding
- Додати підтримку змінної довжини виходу
- Реалізувати режим з ключем
- Створити набір тестів
- Реалізувати benchmark для різних розмірів даних
- Опціонально: реалізувати паралельну версію

6. Результат

Повна реалізація BLAKE2b та BLAKE2s з підтримкою всіх режимів роботи, включаючи MAC режим та змінну довжину виходу. Програма повинна проходити всі тестові вектори та демонструвати високу продуктивність.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
- Модуль констант та параметрів
- Функція змішування G
- Функція стиснення F
- Функції ініціалізації стану
- Основні функції BLAKE2b та BLAKE2s
- Модуль для MAC режиму
- Набір тестів
2. Продемонструвати роботу додатку:
- Правильне хешування тестових векторів
- Робота з різними довжинами виходу
- MAC режим з різними ключами
- Порівняння швидкості з іншими хеш-функціями
- Обробка великих файлів
- Демонстрація паралельної версії (якщо реалізована)
3. Бути готовим відповісти на питання:
- Чому використовуються саме такі константи ротації?
- Як забезпечується безпека при змінній довжині виходу?
- Які оптимізації можливі для конкретної платформи?
- Як BLAKE2 уникає вразливостей інших хеш-функцій?
- Коли варто використовувати BLAKE2 замість SHA-2/3?

Тести

1. Скільки раундів використовує BLAKE2b?
- 1) 10
- 2) 12
- 3) 14
- 4) 16

2. Який розмір блоку в BLAKE2s?
- 1) 64 байти
- 2) 128 байтів

- 3) 256 байтів
- 4) 512 байтів

3. Яка максимальна довжина виходу BLAKE2b?
- 1) 256 біт
- 2) 384 біти
- 3) 512 біт
- 4) 1024 біти

4. На якій конструкції базується BLAKE2?
- 1) Merkle-Damgård
- 2) Sponge
- 3) HAIFA
- 4) Feistel

5. Скільки паралельних екземплярів використовує BLAKE2bp?
- 1) 2
- 2) 4
- 3) 8
- 4) 16

Відповіді:

1-2,

2-1,

3-3,

4-3,

5-2

Тести підвищеної складності

1. Які константи ротації використовуються в функції G для BLAKE2b?
- 1) 16, 12, 8, 7
- 2) 32, 24, 16, 63
- 3) 28, 23, 17, 64
- 4) 32, 25, 16, 11

2. Яка довжина внутрішнього стану BLAKE2b?

- 1) 256 біт
- 2) 512 біт
- 3) 1024 біти
- 4) 2048 біт

3. Що означає прапор фіналізації 0xFFFFFFFFFFFFFFFF?
- 1) Початок повідомлення
- 2) Останній блок
- 3) Помилка обробки
- 4) Режим MAC

4. Скільки слів містить блок повідомлення в BLAKE2b?
- 1) 8
- 2) 12
- 3) 16
- 4) 32

5. Яка основна перевага BLAKE2 над BLAKE?
- 1) Більша безпека
- 2) Менше раундів при тій же безпеці
- 3) Підтримка паралелізму
- 4) Менший розмір коду

6. Що таке tree hashing mode в BLAKE2?
- 1) Режим для хешування файлових систем
- 2) Паралельне хешування великих файлів
- 3) Ієрархічне хешування для верифікації
- 4) Оптимізація для бінарних дерев

7. Яка функція використовується замість padding в BLAKE2?
- 1) Нулі в кінці блоку
- 2) Лічильник довжини
- 3) Спеціальний маркер
- 4) Повторення повідомлення

8. Чому BLAKE2 не потребує HMAC конструкції для MAC?

- 1) Використовує іншу схему padding
- 2) Має вбудовану підтримку ключів
- 3) Стійкий до атак розширення
- 4) Всі відповіді правильні

9. Яка максимальна довжина ключа в BLAKE2s?
- 1) 16 байтів
- 2) 32 байти
- 3) 64 байти
- 4) 128 байтів

10. Що означає персоналізація в BLAKE2?
- 1) Налаштування для конкретного користувача
- 2) Додаткові дані для domain separation
- 3) Ідентифікатор версії
- 4) Криптографічна сіль

Відповіді:

1-2. BLAKE2b використовує константи ротації 32, 24, 16, 63 в функції G

2-2. Внутрішній стан BLAKE2b складається з 8 слів по 64 біти = 512 біт

3-2. Прапор 0xFFFFFFFFFFFFFFFF вказує на останній блок повідомлення

4-3. Блок повідомлення містить 16 слів по 64 біти = 128 байтів

5-2. BLAKE2 використовує 12 раундів замість 14, зберігаючи безпеку

6-2. Tree hashing дозволяє паралельно обробляти великі файли

7-2. BLAKE2 використовує лічильник байтів замість традиційного padding

8-4. Всі три фактори роблять HMAC непотрібним для BLAKE2

9-2. BLAKE2s підтримує ключі до 32 байтів

10-2. Персоналізація дозволяє створювати різні функції для різних застосувань

Завдання для самостійної роботи

1. Реалізуйте BLAKE2x - розширюваний варіант з довільною довжиною виходу більше 512 біт.

2. Дослідіть вплив зменшення кількості раундів на криптографічну стійкість BLAKE2.

3. Реалізуйте оптимізовану версію BLAKE2 з використанням SIMD інструкцій (AVX2/AVX-512).

4. Створіть гібридну схему, що комбінує BLAKE2 з іншою хеш-функцією для підвищення безпеки.

5. Реалізуйте функція виведення ключів (KDF) на основі BLAKE2 згідно з специфікацією.

6. Дослідить можливість використання BLAKE2 для генерації псевдовипадкових чисел.

7. Створіть систему інкрементальне хешування, що дозволяє оновлювати хеш при зміні частини даних.

8. Реалізуйте перевірку цілісності файлової системи з використанням дерева хешів BLAKE2.

9. Порівняйте енергоефективність BLAKE2 з іншими хеш-функціями на мобільних пристроях.

10. Розробіть протокол автентифікації на основі BLAKE2 для IoT мережі.

Теми лабораторних робіт

1. Розробка системи перевірки цілісності файлів з використанням BLAKE2 та Merkle trees.

2. Реалізація високошвидкісної системи дедуплікації даних на основі BLAKE2bp.

3. Створення криптографічного API з підтримкою всіх варіантів BLAKE2.

4. Розробка протоколу безпечного обміну повідомленнями з BLAKE2 MAC.

5. Реалізація блокчейн на основі BLAKE2b.

6. Створення системи паролів з використанням BLAKE2 як KDF.

7. Розробка розподіленої хеш-таблиці з BLAKE2 для адресації.

8. Реалізація системи цифрових підписів з BLAKE2 як хеш-функцією.

9. Створення набору бенчмарків для порівняння хеш-функцій на різних платформах.

Симетричне шифрування AES-GCM

Теоретичні відомості

AES-GCM (Advanced Encryption Standard - Galois/Counter Mode) є одним з найсучасніших та найбезпечніших режимів симетричного шифрування, який поєднує конфіденційність та автентичність даних в одному алгоритмі. Цей режим був стандартизований NIST у 2007 році та став де-факто стандартом для безпечної передачі даних в інтернеті, включаючи протоколи TLS 1.3 та IPsec.

Основні компоненти AES-GCM

AES-GCM складається з двох основних компонентів:

- блокового шифру AES

та

- режиму роботи GCM. Режим GCM забезпечує як шифрування даних (конфіденційність), так і їх автентифікацію (цілісність та автентичність) за допомогою універсального хешування в полі Галуа.

Блоковий шифр AES

AES працює з блоками даних розміром 128 біт та підтримує ключі довжиною 128, 192 або 256 біт. Кількість раундів шифрування залежить від довжини ключа: 10 раундів для 128-бітного ключа, 12 для 192-бітного та 14 для 256-бітного.

Кожен раунд AES складається з чотирьох операцій:

- SubBytes: нелінійна заміна байтів за допомогою S-box
- ShiftRows: циклічний зсув рядків стану
- MixColumns: лінійне перетворення стовпців (крім останнього раунду)
- AddRoundKey: додавання раундового ключа операцією XOR

Режим GCM (Galois/Counter Mode)

GCM поєднує режим лічильника (CTR) для шифрування з універсальним хешуванням для автентифікації. Ключовою особливістю є використання арифметики в полі Галуа $GF(2^{128})$ для обчислення тегу автентифікації.

Компоненти GCM

- Ініціалізаційний вектор (IV): Унікальне значення для кожного повідомлення, зазвичай 96 біт. IV забезпечує семантичну безпеку - однакові повідомлення з різними IV дають різні шифротексти.

- Лічильник: 128-бітне значення, що складається з IV та 32-бітного лічильника блоків. Для 96-бітного IV: Counter = IV || 0^{31} || 1.

- Хеш-ключ H: Обчислюється як H = AES_K(0^{128}), де K - основний ключ шифрування.

Функція GHASH: Універсальна хеш-функція, що працює в полі Галуа GF(2^{128}).

Математичні основи поля Галуа GF(2^{128})

Поле Галуа GF(2^{128}) складається з поліномів степеня менше 128 з коефіцієнтами в GF(2). Операції виконуються за модулем незвідного полінома:

$$p(x) = x^{128} + x^7 + x^2 + x + 1$$

Іншими словами **GF(2^{128})** — це поле з 2^{128} елементів, у якому кожен елемент подається як 128-бітний вектор (або многочлен степеня ⩽127 з коефіцієнтами 0/1).

Основні операції:

1. **Додавання (і віднімання)**

- Виконується поелементно за модулем 2.

- Для 128-бітних слів це просто побітове XOR.

2. **Множення**

- Кожен 128-бітний вектор інтерпретуємо як многочлен $A(x)$ зі ступенем до 127, де біти дають коефіцієнти.

- Виконуємо безпереносне множення многочленів: перемножуємо всі одночлени $x^i \cdot x^j = x^{i+j}$ і складаємо коефіцієнти за модулем 2 (тобто XOR загальних степенів).

- Отриманий многочлен може мати степінь до 254, тому робимо редукцію за фіксованим неприводимим многочленом ступеня 128, наприклад

$$P(x) = x^{128} + x^7 + x^2 + x + 1.$$

результат редукції — многочлен степеня ⩽127, знову 128-бітне слово.

3. **Загальна формула множення**

1. Початково $R(x) = 0$.

2. Для кожного біта a_i в множнику:

- якщо $a_i = 1$, то $R(x) \leftarrow R(x) \oplus A(x) \ll i$ (здвиг на i і XOR).

3. Після завершення зсувів і XOR робимо модульну редукцію $R(x) \bmod P(x)$.

4. **Ділення і обернений елемент**

- Щоб знайти обернений елемент $A^{-1}(x)$, застосовують алгоритм Евкліда для многочленів над GF(2) з тим же неприводимим $P(x)$.

Розглянемо псевдокод виконання множення в GF(2^{128})

```
Функція multiplyGF128(X, Y):
    // X, Y - 128-бітні елементи поля
    // R = 11100001 || 0¹²⁰ (поліном редукції)
    R = 0xE1000000000000000000000000000000

    Z = 0
    V = X

    для і від 0 до 127:
        якщо біт і в Y дорівнює 1:
            Z = Z ⊕ V

        якщо молодший біт V дорівнює 0:
            V = V >> 1  // зсув вправо
        інакше:
            V = (V >> 1) ⊕ R

    повернути Z
```

Алгоритм шифрування AES-GCM

```
Алгоритм AES_GCM_Encrypt(P, K, IV, A):
    // P - відкритий текст
    // K - ключ шифрування
    // IV - ініціалізаційний вектор
    // A - додаткові автентифіковані дані (AAD)

    // Крок 1: Генерація хеш-ключа
    H = AES_K(0¹²⁸)

    // Крок 2: Формування початкового лічильника
    якщо довжина(IV) == 96:
        J₀ = IV || 0³¹ || 1
    інакше:
        J₀ = GHASH(H, IV || 0^s || [len(IV)]₆₄)
```

```
    // Крок 3: Шифрування
    C = GCTR(K, inc₃₂(J₀), P)

    // Крок 4: Обчислення тегу автентифікації
    u = 128 * ⌈len(C)/128⌉ - len(C)
    v = 128 * ⌈len(A)/128⌉ - len(A)

    S = GHASH(H, A || 0^v || C || 0^u || [len(A)]₆₄ || [len(C)]₆₄)
    T = MSB_t(GCTR(K, J₀, S))

    повернути (C, T)
```

Функція GCTR (Counter Mode)

```
Функція GCTR(K, ICB, X):
    // K - ключ
    // ICB - початковий блок лічильника
    // X - дані для обробки

    якщо X порожній:
        повернути порожній рядок

    n = ⌈len(X)/128⌉  // кількість блоків
    CB = ICB
    Y = ""

    для i від 1 до n-1:
        Y = Y || (X[i] ⊕ AES_K(CB))
        CB = inc₃₂(CB)  // інкремент 32-бітного лічильника

    // Останній блок
    Y = Y || (X[n] ⊕ MSB_len(X[n])(AES_K(CB)))

    повернути Y
```

Функція GHASH

```
Функція GHASH(H, X):
    // H - хеш-ключ
    // X - дані для хешування

    m = len(X)/128
    Y₀ = 0¹²⁸

    для i від 1 до m:
        Y_i = (Y_{i-1} ⊕ X_i) • H  // множення в GF(2¹²⁸)

    повернути Y_m
```

Алгоритм розшифрування AES-GCM

```
Алгоритм AES_GCM_Decrypt(C, K, IV, A, T):
    // C - шифротекст
    // T - тег автентифікації
    // Інші параметри як при шифруванні

    // Крок 1: Верифікація тегу
    T' = обчислити тег для (C, K, IV, A)  // як при шифруванні

    якщо T ≠ T':
        повернути ПОМИЛКА_АВТЕНТИФІКАЦІЇ

    // Крок 2: Розшифрування (якщо тег правильний)
    P = GCTR(K, inc₃₂(J₀), C)

    повернути P
```

Розглянемо шифрування короткого повідомлення з використанням AES-128-GCM.

Вхідні дані:

- Ключ K: 0x00000000000000000000000000000000 (128 біт)

- IV: 0x000000000000000000000000 (96 біт)

- Відкритий текст P: 0x00000000000000000000000000000000 (128 біт)

- AAD A: порожній

Крок 1: Генерація хеш-ключа

$H = AES\_K(0^{128}) = 0x66e94bd4ef8a2c3b884cfa59ca342b2e$

Крок 2: Формування початкового лічильника

Оскільки IV має довжину 96 біт:

$J_0 = IV \,||\, 0^{31} \,||\, 1 = 0x00000000000000000000000000000001$

Крок 3: Шифрування

$$CB_1 = inc_{32}(J_0) = 0x00000000000000000000000000000002$$
$$C = P \oplus AES\_K(CB_1) = 0x00000000000000000000000000000000 \oplus$$
$$0x0388dace60b6a392f328c2b971b2fe78$$
$$= 0x0388dace60b6a392f328c2b971b2fe78$$

Крок 4: Обчислення тегу автентифікації

Формуємо вхід для GHASH:

$$S\_input = A \,||\, 0^v \,||\, C \,||\, 0^u \,||\, [len(A)]_{64} \,||\, [len(C)]_{64}$$
$$= C \,||\, 0x00000000000000000000000000000080$$

Обчислюємо GHASH:

$Y_1 = (0 \oplus C) \cdot H = C \cdot H$ в $GF(2^{128})$
$Y_2 = (Y_1 \oplus 0x00000000000000000000000000000080) \cdot H$
$S = Y_2$

Обчислюємо тег:

$T = AES\_K(J_0) \oplus S = 0x58e2fccefa7e3061367f1d57a4e7455a \oplus S$

Оптимізації та реалізаційні аспекти

1. Передобчислення для GHASH

Для прискорення множення в $GF(2^{128})$ можна передобчислити степені H:

$H^2 = H \cdot H$
$H^3 = H^2 \cdot H$
$H^4 = H^3 \cdot H$
...

Це дозволяє обробляти декілька блоків паралельно.

2. Використання інструкцій процесора

Сучасні процесори мають спеціальні інструкції для GCM: - Intel: PCLMULQDQ для множення в $GF(2^{128})$ - ARM: PMULL для поліноміального множення - AES-NI: апаратне прискорення AES

3. Вимоги безпеки

Унікальність IV: Кожен IV повинен використовуватися тільки один раз з одним ключем. Повторне використання IV компрометує як конфіденційність, так і автентичність.

Обмеження на довжину даних: Максимум 2^{39} - 256 біт відкритого тексту та 2^{64} - 1 біт AAD для одного ключа.

Довжина тегу: Рекомендується використовувати повний 128-бітний тег. Коротші теги знижують безпеку автентифікації.

Область застосування

AES-GCM широко використовується в сучасних криптографічних протоколах та системах:

- Мережеві протоколи: TLS 1.2/1.3 використовує AES-GCM як основний шифронабір. IPsec підтримує AES-GCM для ESP (Encapsulating Security Payload). SSH включає AES-GCM для безпечних з'єднань.

- Хмарні сховища: AWS, Google Cloud та Azure використовують AES-GCM для шифрування даних at rest та in transit. Забезпечує як конфіденційність, так і перевірку цілісності даних.

- Мобільні платформи: Android та iOS використовують AES-GCM для шифрування файлової системи та безпечного зберігання ключів. WhatsApp, Signal та інші месенджери використовують AES-GCM в протоколі Signal.

- Веб-додатки: Web Crypto API підтримує AES-GCM для клієнтського шифрування. JSON Web Encryption (JWE) використовує AES-GCM як рекомендований алгоритм.

- Віртуалізація: Гіпервізори використовують AES-GCM для шифрування пам'яті віртуальних машин. Intel SGX використовує AES-GCM для захисту енклавів.

Приклади застосування

Приклад 1: Безпечний обмін повідомленнями

Месенджер використовує AES-GCM для end-to-end шифрування. Кожне повідомлення шифрується унікальним сесійним ключем, отриманим через протокол Діффі-Геллмана. IV генерується з лічильника повідомлень та випадкової компоненти. Метадані повідомлення (відправник, час) включаються як AAD, забезпечуючи їх автентичність без розкриття змісту.

Приклад 2: Захист API запитів

REST API використовує AES-GCM для захисту чутливих даних. Клієнт шифрує тіло запиту, включаючи URL та заголовки як AAD. Сервер перевіряє автентичність всього запиту перед обробкою. Це захищає від атак replay та tampering, забезпечуючи цілісність всього HTTP запиту.

Приклад 3: Шифрування бази даних

Система управління базами даних використовує AES-GCM для прозорого шифрування. Кожен рядок шифрується з унікальним IV, похідним від первинного ключа. Індекси зберігаються як AAD, дозволяючи пошук без розшифрування. Тег автентифікації захищає від несанкціонованих змін даних.

Переваги та недоліки

Переваги

- Швидкість: AES-GCM один з найшвидших authenticated encryption алгоритмів. На сучасних процесорах з AES-NI та PCLMULQDQ досягає швидкості понад 10 Гбіт/с.

- Безпека: Забезпечує як конфіденційність (IND-CPA), так і автентичність (INT-CTXT). Математично доведена безпека при правильному використанні.

- Паралелізм: Як шифрування, так і автентифікація можуть виконуватися паралельно. Дозволяє ефективно використовувати багатоядерні процесори.

- Стандартизація: Включений в усі основні криптографічні стандарти. Широка підтримка в апаратному та програмному забезпеченні.

- Однопрохідність: Обробляє дані за один прохід, на відміну від encrypt-then-MAC схем.

Недоліки

- Вразливість до повторного використання IV: Повторне використання IV з одним ключем повністю компрометує безпеку. Дозволяє відновити хеш-ключ H та підробляти повідомлення.

- Відсутність стійкості до неправильного використання: На відміну від SIV або GCM-SIV, не забезпечує мінімальної безпеки при повторенні IV.

- Складність реалізації: Правильна реалізація арифметики в $GF(2^{128})$ нетривіальна.

- Обмеження на довжину: Обмеження 2^{39} - 256 біт на повідомлення може бути проблемою для деяких застосувань.

- Відсутність комутативності: Порядок блоків важливий, що ускладнює деякі сценарії використання.

Контрольні запитання

1. Які два основні компоненти забезпечують безпеку в AES-GCM?

2. Чому важливо ніколи не повторювати IV з одним ключем?

3. Яка роль хеш-ключа H в алгоритмі GCM?

4. Як працює лічильник в режимі CTR?

5. Що таке поле Галуа $GF(2^{128})$ і чому воно використовується?

6. Яка різниця між AAD та шифрованими даними?

7. Як обчислюється тег автентифікації?

8. Чому AES-GCM швидший за послідовне застосування AES-CBC та HMAC?

9. Які апаратні інструкції прискорюють AES-GCM?

10. Що відбувається при спробі розшифрування з неправильним тегом?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму симетричного шифрування AES-GCM

2. Мета

Вивчити принципи автентифіковане шифрування, реалізувати алгоритм AES-GCM, дослідити його властивості безпеки та продуктивність.

3. Завдання

- Реалізувати арифметику в полі Галуа GF(2^{128})
- Реалізувати функцію GHASH
- Реалізувати режим лічильника GCTR
- Об’єднати компоненти в повний алгоритм AES-GCM
- Реалізувати генерацію та перевірку тегів автентифікації
- Додати підтримку AAD
- Провести тестування на векторах NIST
- Дослідити вплив повторного використання IV

4. Опис

Лабораторна робота передбачає покрокову реалізацію AES-GCM від базових математичних операцій до повного алгоритму. Особлива увага приділяється правильній реалізації арифметики поля Галуа та забезпеченню криптографічної стійкості.

5. Порядок виконання

- Реалізувати множення в GF(2^{128}) з редукцією
- Реалізувати блоковий шифр AES (або використати готову реалізацію)
- Реалізувати функцію GHASH з правильним padding
- Реалізувати режим лічильника з інкрементом
- Об’єднати компоненти згідно специфікації
- Реалізувати обробку IV різної довжини
- Додати підтримку часткових тегів
- Створити набір тестів
- Провести аналіз продуктивності
- Продемонструвати атаку при повторенні IV

6. Результат

Повнофункціональна реалізація AES-GCM з підтримкою всіх особливостей стандарту. Програма повинна проходити всі тестові вектори NIST та демонструвати практичне застосування для шифрування файлів.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Модуль арифметики GF(2^{128})
- Реалізація GHASH
- Реалізація GCTR
- Функції шифрування та розшифрування AES-GCM
- Модуль роботи з IV та лічильниками
- Набір тестів NIST

2. Продемонструвати роботу додатку:

- Шифрування та розшифрування з перевіркою автентичності
- Робота з AAD
- Обробка помилок автентифікації
- Демонстрація атаки при повторенні IV
- Порівняння швидкості з AES-CBC + HMAC
- Шифрування великих файлів

3. Бути готовим відповісти на питання:

- Як забезпечується унікальність лічильників?
- Чому множення в GF(2^{128}) безпечне для хешування?
- Як оптимізувати GHASH для довгих повідомлень?
- Які загрози несе повторне використання IV?
- Коли використовувати AES-GCM замість інших режимів?

Тести

1. Який розмір блоку використовує AES?

- 1) 64 біти
- 2) 128 біт
- 3) 256 біт

- 4) 512 біт

2. Яка максимальна рекомендована довжина IV для AES-GCM?

- 1) 64 біти

- 2) 96 біт

- 3) 128 біт

- 4) 256 біт

3. Що означає AAD в контексті AES-GCM?

- 1) Advanced Authentication Data

- 2) Additional Authenticated Data

- 3) Associated Authentication Digest

- 4) Auxiliary Authenticated Data

4. В якому полі виконується універсальне хешування в GCM?

- 1) GF(2^{64})

- 2) GF(2^{128})

- 3) GF(2^{256})

- 4) GF(p) для простого p

5. Що відбувається при невідповідності тегу автентифікації?

- 1) Повертається частково розшифрований текст

- 2) Повертається повідомлення про помилку

- 3) Ігнорується і продовжується розшифрування

- 4) Автоматично виправляється тег

Відповіді:

1-2,

2-2,

3-2,

4-2,

5-2

Тести підвищеної складності

1. Який поліном використовується для редукції в GF(2^{128})?

- 1) $x^{128} + x^7 + x^2 + x + 1$

- 2) $x^{128} + x^{64} + x^{32} + x + 1$

- 3) $x^{128} + x^{127} + x^{126} + x + 1$

- 4) $x^{128} + 1$

2. Скільки разів викликається AES для шифрування n блоків в GCM?

- 1) n

- 2) n + 1

- 3) n + 2

- 4) 2n

3. Яка складність найкращої відомої атаки на GCM при правильному використанні?

- 1) 2^{64}

- 2) 2^{128}

- 3) 2^{256}

- 4) Залежить від довжини тегу

4. Що таке forbidden attack в контексті GCM?

- 1) Атака на слабкі ключі

- 2) Відновлення H при повторенні IV

- 3) Атака на коротші теги

- 4) Side-channel атака

5. Який розмір внутрішнього стану лічильника в GCM?

- 1) 32 біти

- 2) 64 біти

- 3) 96 біт

- 4) 128 біт

6. Як називається властивість, яку забезпечує authenticated encryption?

- 1) IND-CPA

- 2) IND-CCA2

- 3) INT-CTXT

- 4) IND-CCA2 та INT-CTXT

7. Чому GCM використовує little-endian для лічильника?

- 1) Для сумісності з Intel
- 2) Історичні причини
- 3) Для ефективного інкременту
- 4) Вимога стандарту AES

8. Яка максимальна кількість блоків для одного IV?
- 1) 2^{16}
- 2) 2^{32}
- 3) 2^{64}
- 4) 2^{128}

9. Що таке GCM-SIV?
- 1) Версія GCM для коротких повідомлень
- 2) GCM з синтетичним IV
- 3) GCM з подвійним шифруванням
- 4) GCM для streaming

10. Яка операція НЕ використовується в GHASH?
- 1) XOR
- 2) Множення в GF(2^{128})
- 3) Зсув
- 4) Додавання за модулем 2^{128}

Відповіді:

1-1. Стандартний поліном редукції для GCM: $x^{128} + x^7 + x^2 + x + 1$

2-3. n блоків даних + 1 для H + 1 для тегу = n + 2

3-2. При правильному використанні безпека визначається розміром блоку AES

4-2. **Forbidden attack** дозволяє відновити H при повторенні nonce

5-4. Лічильник має 128 біт: 96 біт IV + 32 біти counter

6-4. AE забезпечує і конфіденційність (IND-CCA2) і цілісність (INT-CTXT)

7-3. **Little-endian** дозволяє ефективний інкремент молодших 32 біт

8-2. Обмежено 32-бітним лічильником: 2^{32} блоки

9-2. GCM-SIV використовує синтетичний IV для стійкості до неправильного використання

10-4. GHASH використовує тільки XOR та множення в полі Галуа

Завдання для самостійної роботи

1. Реалізуйте оптимізовану версію GHASH з використанням методу Карацуби для множення в $GF(2^{128})$.

2. Дослідіть та реалізуйте атаку відновлення хеш-ключа H при повторному використанні IV.

3. Створіть версію AES-GCM з підтримкою Інкрементальної автентифікації для потокової обробки даних.

4. Реалізуйте версію множення в $GF(2^{128})$ для захисту від timing атак за константний час.

5. Розробіть систему **key wrapping** на основі AES-GCM для безпечного зберігання ключів.

6. Порівняйте продуктивність AES-GCM з ChaCha20-Poly1305 на різних платформах.

7. Реалізуйте AES-GCM з використанням SIMD інструкцій (AES-NI, PCLMULQDQ).

8. Створіть систему для виявлення повторного використання IV в реальному часі.

9. Дослідіть вплив довжини тегу на безпеку та розробіть рекомендації для різних сценаріїв.

10. Реалізуйте протокол безпечного оновлення ключів для довготривалих з’єднань з AES-GCM.

Теми лабораторних робіт

1. Розробка безпечного протоколу обміну файлами з використанням AES-GCM та ECDH.

2. Реалізація VPN клієнта з підтримкою AES-GCM для шифрування трафіку.

3. Створення системи шифрування бази даних з використанням AES-GCM для кожного запису.

4. Розробка безпечного чату з **end-to-end** шифруванням на основі AES-GCM.

5. Реалізація системи захисту API з автентифікацією запитів через AES-GCM.

6. Створення менеджера паролів з шифруванням сховища через AES-GCM.

7. Розробка системи безпечного зберігання файлів в хмарі з клієнтським шифруванням.

8. Реалізація протоколу безпечного оновлення прошивки IoT пристроїв.

9. Створення системи логування з криптографічним захистом цілісності логів.

10. Розробка бібліотеки для безпечної серіалізації об'єктів з автоматичним шифруванням.

12.3. Алгебраїчні алгоритми

Алгебраїчні алгоритми становлять фундаментальну основу сучасної обчислювальної математики та комп'ютерних наук. Ці алгоритми вирішують базові математичні задачі, які на перший погляд можуть здаватися простими, але їх ефективна реалізація має критичне значення для продуктивності складних систем. Від криптографічних протоколів до обробки сигналів, від комп'ютерної графіки до машинного навчання - скрізь використовуються алгебраїчні обчислення, швидкість яких безпосередньо впливає на загальну ефективність системи.

У цьому розділі ми розглянемо три класичні алгебраїчні алгоритми, які демонструють, як математична елегантність поєднується з практичною ефективністю. Кожен з них вирішує задачу, для якої існує очевидний прямолінійний підхід, але завдяки глибокому математичному аналізу вдається досягти значно кращої асимптотичної складності.

Швидке піднесення до степеня показує, як задачу обчислення a^n можна вирішити за $O(log\, n)$ операцій замість наївних $O(n)$. Цей алгоритм є наріжним каменем сучасної криптографії, де часто доводиться обчислювати степені з показниками в сотні або тисячі біт. Ми розглянемо як базовий бінарний метод, так і більш складні оптимізації, включаючи метод sliding window та алгоритми для спеціальних випадків.

Швидке перетворення Фур'є (FFT) революціонізувало цифрову обробку сигналів, зменшивши складність дискретного перетворення Фур'є з $O(n^2)$ до $O(n\, log\, n)$. Алгоритм Кулі-Тьюкі, який ми детально вивчимо, базується

на принципі “розділяй та володарюй” та властивостях комплексних коренів з одиниці. FFT не тільки дозволяє ефективно аналізувати частотний спектр сигналів, але й лежить в основі швидкого множення великих чисел та поліномів.

Алгоритм Штрассена для множення матриць демонструє неочевидний підхід до оптимізації: замість зменшення кількості арифметичних операцій на елементарному рівні, він змінює саму структуру обчислень. Зменшуючи кількість рекурсивних множень з 8 до 7 для матриць 2×2, Штрассен досягає складності $O(n^{2.807})$ замість стандартних $O(n^3)$. Цей алгоритм став першим проривом у довгій історії пошуку оптимальних алгоритмів множення матриць.

Вивчення цих алгоритмів дає не тільки конкретні інструменти для вирішення практичних задач, але й глибоке розуміння принципів алгоритмічної оптимізації.

Кожен алгоритм буде представлений з повним математичним обґрунтуванням, детальним псевдокодом, аналізом складності та практичними прикладами застосування. Особлива увага приділятиметься не тільки асимптотичній складності, але й константам, які часто визначають практичну ефективність алгоритму для реальних розмірів задач.

Швидке піднесення до степеня

Теоретичні відомості

Швидке піднесення до степеня є фундаментальним алгоритмом в обчислювальній математиці, який дозволяє обчислити значення a^n за логарифмічну кількість операцій замість лінійної. Цей алгоритм має критичне значення для багатьох областей комп’ютерних наук, від криптографії до обчислювальної геометрії, де часто виникає необхідність піднесення чисел до великих степенів.

Основна ідея алгоритму

Наївний підхід до обчислення a^n полягає в послідовному множенні числа a на себе n разів, що вимагає $n-1$ операцій множення. Однак, використовуючи властивості степенів та двійкове представлення показника степеня, можна значно зменшити кількість необхідних операцій.

Ключова ідея базується на наступних спостереженнях:

- Якщо n парне: $a^n = (a^{(\frac{n}{2})})^2$

- Якщо n непарне: $a^n = a \times a^{(n-1)} = a \times (a^{(\frac{n-1}{2})}))^2$

Це дозволяє рекурсивно зводити задачу обчислення a^n до задачі обчислення $a^{(\frac{n}{2})}$, зменшуючи показник степеня вдвічі на кожному кроці.

Математичне обґрунтування

Розглянемо двійкове представлення числа n. Нехай

$$n = b_k \times 2^k + b_(k-1) \times 2^(k-1) + \ldots + b_1 \times 2 + b_0, \text{ де } b_i \in 0,1$$

.

Тоді:

$$\begin{aligned} a^n &= a^(b_k \times 2^k + b_(k-1) \times 2^(k-1) + \ldots + b_1 \times 2 + b_0) \\ &= a^(b_k \times 2^k) \times a^(b_(k-1) \times 2^(k-1)) \times \ldots \times a^(b_1 \times 2) \times a_0^b \\ &= (a^(2^k))_k^b \times (a^(2^(k-1)))_(k-1)^b \times \ldots \times (a^2)_1^b \times a_0^b \end{aligned}$$

Оскільки b_i може бути лише 0 або 1, кожен множник або дорівнює 1 (якщо $b_i = 0$), або дорівнює відповідному степеню a (якщо $b_i = 1$).

Базовий рекурсивний алгоритм

```
Функція швидкеПіднесенняРекурсивне(a, n):
    // Базовий випадок
    якщо n == 0:
        повернути 1

    // Рекурсивний випадок
    якщо n парне:
        половина = швидкеПіднесенняРекурсивне(a, n / 2)
        повернути половина × половина
    інакше:
        половина = швидкеПіднесенняРекурсивне(a, (n - 1) / 2)
        повернути a × половина × половина
```

Ітеративний алгоритм (бінарне піднесення)

Ітеративна версія алгоритму часто є більш ефективною, оскільки уникає накладних витрат на рекурсивні виклики:

```
Функція швидкеПіднесенняІтеративне(a, n):
    результат = 1
    основа = a
    показник = n
```

```
    поки показник > 0:
        // Якщо поточний біт показника дорівнює 1
        якщо показник mod 2 == 1:
            результат = результат × основа

        // Підготовка до наступного біта
        основа = основа × основа
        показник = показник div 2

    повернути результат
```

Оптимізований алгоритм з обробкою від'ємних степенів

```
Функція швидкеПіднесенняПовне(a, n):
    // Обробка особливих випадків
    якщо a == 0:
        якщо n == 0:
            повернути невизначено  // 0^0 математично не визначено
        якщо n < 0:
            повернути нескінченність  // ділення на нуль
        повернути 0

    якщо n == 0:
        повернути 1

    // Обробка від'ємних степенів
    якщо n < 0:
        повернути 1 / швидкеПіднесенняПовне(a, -n)

    // Основний алгоритм для додатних n
    результат = 1
    основа = a
    показник = n

    поки показник > 0:
        якщо показник mod 2 == 1:
            результат = результат × основа
        основа = основа × основа
        показник = показник div 2

    повернути результат
```

Модульне піднесення до степеня

У криптографії часто потрібно обчислювати $a^n mod m$ для великих значень. Пряме обчислення a^n з подальшим взяттям модуля неможливе через

переповнення. Алгоритм модульного піднесення до степеня вирішує цю проблему:

```
Функція модульнеПіднесення(a, n, m):
    // Нормалізація основи
    a = a mod m

    // Особливі випадки
    якщо m == 1:
        повернути 0
    якщо n == 0:
        повернути 1

    результат = 1
    основа = a
    показник = n

    поки показник > 0:
        якщо показник mod 2 == 1:
            результат = (результат × основа) mod m
        основа = (основа × основа) mod m
        показник = показник div 2

    повернути результат
```

Метод вікна

Для подальшої оптимізації можна використовувати метод вікна, який обробляє декілька бітів показника одночасно:

```
Функція піднесенняМетодомВікна(a, n, w):
    // w - розмір вікна (зазвичай 3-6)
    // Передобчислення степенів a
    передобчислені = масив[2^w]
    передобчислені[0] = 1
    передобчислені[1] = a

    для і від 2 до 2^w - 1:
        передобчислені[і] = передобчислені[і-1] × a

    // Розбиття n на блоки по w бітів
    результат = 1
    біти = двійковеПредставлення(n)
    і = довжина(біти) - 1

    поки і >= 0:
        // Зчитування w бітів
```

```
        вікно = 0
        для j від 0 до min(w-1, i):
            якщо біти[i-j] == 1:
                вікно = вікно + 2^j

        // Обробка вікна
        для j від 0 до min(w-1, i):
            результат = результат × результат

        якщо вікно != 0:
            результат = результат × передобчислені[вікно]

        i = i - w

    повернути результат
```

Обчислимо 3^{13} використовуючи швидке піднесення до степеня.

Спочатку представимо 13 у двійковій системі: $13 = 1101_2 = 8 + 4 + 1$

Метод 1: Рекурсивний підхід

$3^{13} = 3 \times 3^{12}$ $3^{12} = (3^6)^2$ $3^6 = (3^3)^2$ $3^3 = 3 \times 3^2$ $3^2 = (3^1)^2$ $3^1 = 3$

Обчислення знизу вгору: - $3^1 = 3$ - $3^2 = 3 \times 3 = 9$ - $3^3 = 3 \times 9 = 27$ - $3^6 = 27 \times 27 = 729$ - $3^{12} = 729 \times 729 = 531441$ - $3^{13} = 3 \times 531441 = 1594323$

Метод 2: Ітеративний підхід

Крок	Показник	Біт	Основа	Результат
0	$13 = 1101_2$	-	3	1
1	13	1	3	$1 \times 3 = 3$
2	$6 = 110_2$	0	9	3
3	$3 = 11_2$	1	81	$3 \times 81 = 243$
4	$1 = 1_2$	1	6561	$243 \times 6561 = 1594323$

Покроковий процес:

1. Початково: результат = 1, основа = 3, показник = 13

2. 13 непарне, тому результат = $1 \times 3 = 3$

3. Оновлюємо: основа = $3 \times 3 = 9$, показник = $13 \div 2 = 6$

4. 6 парне, тому результат залишається 3

5. Оновлюємо: основа = $9 \times 9 = 81$, показник = $6 \div 2 = 3$

6. 3 непарне, тому результат = $3 \times 81 = 243$

7. Оновлюємо: основа = 81 × 81 = 6561, показник = 3 ÷ 2 = 1

8. 1 непарне, тому результат = 243 × 6561 = 1594323

9. Показник стає 0, алгоритм завершується

Аналіз складності

Часова складність: $O(logn)$, де n - показник степеня. Це випливає з того, що на кожній ітерації показник зменшується вдвічі.

Просторова складність:

- Рекурсивна версія: O(log n) через стек викликів

- Ітеративна версія: O(1) - використовується лише константна кількість змінних

Кількість операцій множення:

- Найгірший випадок: 2⌊$log_2 n$⌋ множень (коли всі біти n дорівнюють 1)

- Середній випадок: 1.5⌊$log_2 n$⌋ множень

- Найкращий випадок: ⌊$log_2 n$⌋ множень (коли n є степенем двійки)

Узагальнення для інших операцій

Алгоритм швидкого піднесення до степеня можна узагальнити для будь-якої асоціативної операції. Наприклад:

```
Функція швидкеЗастосуванняОперації(a, n, операція, нейтральнийЕлемент
):
    результат = нейтральнийЕлемент
    основа = a
    показник = n

    поки показник > 0:
        якщо показник mod 2 == 1:
            результат = операція(результат, основа)
        основа = операція(основа, основа)
        показник = показник div 2

    повернути результат
```

Це дозволяє використовувати той самий алгоритм для: - Множення матриць: A^n - Композиції функцій: $f \circ f \circ ... \circ f$ (n разів) - Модульного множення в кільцях/

Область застосування

Швидке піднесення до степеня має широке застосування в різних галузях комп'ютерних наук та математики:

- Криптографія з відкритим ключем: RSA, DSA, еліптичні криві - всі ці системи інтенсивно використовують модульне піднесення до степеня з великими показниками (сотні або тисячі біт).

- Тестування простоти чисел: Тест Міллера-Рабіна та інші ймовірнісні тести простоти використовують модульне піднесення до степеня для перевірки властивостей чисел.

- Обчислювальна теорія чисел: Обчислення символів Лежандра та Якобі, знаходження примітивних коренів, розв'язання діофантових рівнянь.

- Комп'ютерна алгебра: Операції в скінченних полях, обчислення в кільцях лишків, робота з поліномами.

- Чисельні методи: Швидке обчислення членів рекурентних послідовностей, степеневі методи для власних значень матриць.

- Комбінаторика та динамічне програмування: Обчислення великих біноміальних коефіцієнтів, підрахунок кількості шляхів у графах.

Приклади застосування

Приклад 1: RSA шифрування

У системі RSA повідомлення M шифрується як $C = M^e mod n$, де e - відкритий ключ, n - модуль. Для типових розмірів ключів (2048 біт) пряме обчислення M^e неможливе. Використання швидкого модульного піднесення до степеня дозволяє виконати шифрування за долі секунди:

```
// Приклад з малими числами для демонстрації
n = 3233, e = 17, M = 65
C = модульнеПіднесення(65, 17, 3233) = 2790
```

Приклад 2: Обчислення чисел Фібоначчі

N-те число Фібоначчі можна обчислити через піднесення матриці до степеня:

```
|F(n+1) F(n)  | = |1 1|^n
|F(n)   F(n-1)|   |1 0|
```

Використовуючи швидке піднесення матриці до степеня, можна знайти $F(n)$ за O(log n) операцій замість O(n).

Приклад 3: Підрахунок шляхів у графі

Кількість шляхів довжини k між вершинами i та j у графі дорівнює елементу (i,j) матриці суміжності, піднесеної до степеня k. Швидке піднесення до степеня дозволяє ефективно знаходити кількість шляхів для великих k.

Переваги та недоліки

Переваги

- Логарифмічна складність: Зменшення кількості операцій з $O(n)$ до $O(log\ n)$ дає колосальний виграш для великих показників степеня.

- Універсальність: Алгоритм працює для будь-якої асоціативної операції, не тільки для множення чисел.

- Простота реалізації: Базова версія алгоритму легко реалізується та не вимагає складних структур даних.

- Ефективність для модульної арифметики: Дозволяє уникнути переповнення при роботі з великими числами.

- Можливість оптимізації: Існують різні варіанти оптимізації для конкретних випадків використання.

Недоліки

- Накладні витрати для малих степенів: Для малих n (менше 10-20) простий алгоритм може бути швидшим через менші константи.

- Непридатність для паралелізації: Кожен крок залежить від попереднього, що ускладнює паралельну реалізацію.

- Чутливість до розміру операндів: При роботі з великими числами час однієї операції множення може домінувати над кількістю операцій.

- Складність оптимальної реалізації: Вибір оптимального розміру вікна та інших параметрів залежить від конкретної архітектури та розміру чисел.

Контрольні запитання

1. Яка основна ідея алгоритму швидкого піднесення до степеня?

2. Чому алгоритм має логарифмічну складність?

3. Яка різниця між рекурсивною та ітеративною реалізаціями?

4. Як працює модульне піднесення до степеня і чому воно важливе?

5. Що таке метод вікна і коли його варто використовувати?

6. Як узагальнити алгоритм для інших асоціативних операцій?

7. Скільки множень потрібно для обчислення a^15 у найгіршому випадку?

8. Чому для криптографії критично важливе модульне піднесення до степеня?

9. Як обробляються від’ємні показники степеня?

10. Які обмеження має алгоритм швидкого піднесення до степеня?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритмів швидкого піднесення до степеня

2. Мета

Вивчити принципи швидкого піднесення до степеня, реалізувати різні варіанти алгоритму, дослідити їх ефективність та застосувати для розв’язання практичних задач.

3. Завдання

- Реалізувати рекурсивну версію швидкого піднесення до степеня
- Реалізувати ітеративну версію алгоритму
- Реалізувати модульне піднесення до степеня
- Реалізувати метод вікна з різними розмірами вікна
- Реалізувати піднесення матриць до степеня
- Порівняти продуктивність різних реалізацій
- Застосувати алгоритми для обчислення чисел Фібоначчі
- Реалізувати просту версію RSA шифрування

4. Опис

Лабораторна робота передбачає поетапну реалізацію різних варіантів алгоритму швидкого піднесення до степеня, від базових версій до оптимізованих реалізацій. Особлива увага приділяється правильній обробці граничних випадків та ефективності для різних розмірів вхідних даних.

5. Порядок виконання

- Реалізувати допоміжні функції для роботи з великими числами
- Реалізувати та протестувати рекурсивну версію алгоритму
- Реалізувати ітеративну версію та порівняти з рекурсивною
- Додати підтримку від’ємних показників

- Реалізувати модульне піднесення до степеня
- Реалізувати метод вікна з передобчисленням
- Адаптувати алгоритм для множення матриць
- Створити тестовий набір для перевірки коректності
- Провести вимірювання продуктивності
- Реалізувати практичні застосування

6. Результат

Набір функцій для швидкого піднесення до степеня з підтримкою різних типів даних та операцій. Програма повинна демонструвати коректність роботи на тестових прикладах та показувати виграш у продуктивності порівняно з наївним алгоритмом.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
- Рекурсивна реалізація алгоритму
- Ітеративна реалізація алгоритму
- Модульне піднесення до степеня
- Реалізація методу вікна
- Функції для роботи з матрицями
- Тестові функції та benchmarks

2. Продемонструвати роботу додатку:
- Обчислення великих степенів чисел
- Порівняння часу роботи різних алгоритмів
- Роботу модульного піднесення до степеня
- Обчислення чисел Фібоначчі через матриці
- Просте RSA шифрування/розшифрування
- Графіки залежності часу від розміру показника

3. Бути готовим відповісти на питання:
- Чому ітеративна версія зазвичай швидша за рекурсивну?
- Як вибрати оптимальний розмір вікна?
- Які переваги дає модульна арифметика?

- Як адаптувати алгоритм для інших операцій?
- Які існують альтернативні методи оптимізації?

Тести

1. Яка часова складність алгоритму швидкого піднесення до степеня?
- 1) O(n)
- 2) O(log n)
- 3) O(n log n)
- 4) $O(n^2)$

2. Скільки множень потрібно для обчислення a^16 швидким алгоритмом?
- 1) 4
- 2) 5
- 3) 8
- 4) 16

3. Яке твердження правильне для модульного піднесення до степеня?
- 1) (a^n) mod m = ((a mod m)^n) mod m
- 2) Потрібно спочатку обчислити a^n, потім взяти модуль
- 3) Модуль береться на кожному кроці алгоритму
- 4) Модульне піднесення неможливе для великих n

4. Що є основою методу вікна?
- 1) Обробка одного біта за раз
- 2) Обробка декількох бітів одночасно
- 3) Використання рекурсії
- 4) Паралельні обчислення

5. Для якої операції НЕ можна застосувати швидке піднесення до степеня?
- 1) Множення матриць
- 2) Композиція функцій
- 3) Додавання чисел
- 4) Множення в скінченному полі

Відповіді:

1-2,

2-1,

3-3,

4-2,

5-3

Тести підвищеної складності

1. Яка мінімальна кількість множень для обчислення a^15?

- 1) 5
- 2) 6
- 3) 7
- 4) 8

2. Який метод дає найкращу асимптотику для дуже великих показників?

- 1) Бінарний метод
- 2) Метод вікна
- 3) k-арний метод
- 4) Всі мають однакову асимптотику

3. Чому в криптографії важливо використовувати **constant-time** реалізації?

- 1) Для швидкості
- 2) Для захисту від **timing attacks**
- 3) Для економії пам’яті
- 4) Для точності обчислень

4. Яка оптимальна довжина вікна для 1024-бітних показників?

- 1) 1-2 біти
- 2) 3-4 біти
- 3) 5-6 бітів
- 4) 7-8 бітів

5. Що таке додаткові ланцюжки?

- 1) Спосіб паралелізації
- 2) Оптимальна послідовність операцій
- 3) Метод кешування
- 4) Тип рекурсії

6. Яка складність знаходження оптимального додаткового ланцюжка?

- 1) O(log n)

- 2) O(n)

- 3) NP-повна задача

- 4) $O(n^2)$

7. Для якого випадку редукція Монгомері дає переваги?

- 1) Малі числа

- 2) Степені двійки

- 3) Багаторазове модульне множення

- 4) Від'ємні показники

8. Що таке **sliding window** метод?

- 1) Фіксоване вікно

- 2) Змінний розмір вікна

- 3) Пропуск нульових бітів

- 4) Паралельне вікно

9. Яка мінімальна глибина для обчислення a^n паралельним алгоритмом?

- 1) O(1)

- 2) O(log log n)

- 3) O(log n)

- 4) $O(\sqrt{n})$

10. Що є основним обмеженням паралелізації швидкого піднесення до степеня?

- 1) Розмір пам'яті

- 2) Послідовні залежності

- 3) Складність синхронізації

- 4) Неможливість розпаралелювання

Відповіді:

1-2. Для a^15 = a^(1111_2) потрібно 6 множень використовуючи додатковий ланцюжок

2-4. Всі методи мають O(log n) складність, відрізняються лише константами

3-2. **Constant-time** реалізації запобігають витоку інформації через час виконання

4-3. Для 1024-бітних чисел оптимальне вікно 5-6 бітів балансує передобчислення та швидкість

5-2. Додаткові ланцюги - це оптимальні послідовності додавань для досягнення числа

6-3. Знаходження найкоротшого додаткового ланцюжка є NP-повною задачею

7-3. Редукція Монтгомері ефективна при багаторазових модульних оп3раціях

8-3. **Sliding window** пропускає послідовності нулів, обробляючи лише значущі біти

9-2. Паралельний алгоритм може досягти $O(log\ log\ n)$ глибини

10-2. Кожен крок залежить від попереднього, що обмежує паралелізм

Завдання для самостійної роботи

1. Реалізуйте алгоритм з використанням редукції Монтгомері для ефективного модульного піднесення до степеня.

2. Дослідіть та реалізуйте метод **sliding window** і порівняйте його ефективність з фіксованим вікном.

3. Створіть паралельну версію алгоритму для обчислення декількох незалежних піднесень до степеня.

4. Реалізуйте швидке піднесення до степеня для поліномів над скінченним полем.

5. Розробіть алгоритм для одночасного обчислення a^m та a^n з мінімальною кількістю операцій.

6. Дослідіть вплив різних представлень показника (NAF, двійкове, тернарне) на ефективність.

7. Реалізуйте **constant-time** версію алгоритму для криптографічних застосувань.

8. Створіть адаптивний алгоритм, що вибирає оптимальний метод залежно від розміру показника.

9. Розробіть візуалізацію процесу швидкого піднесення до степеня для навчальних цілей.

10. Дослідіть та реалізуйте алгоритми для обчислення a^n mod m, коли m - складене число з відомою факторизацією.

Теми лабораторних робіт

1. Реалізація та порівняння різних алгоритмів швидкого піднесення до степеня.

2. Розробка бібліотеки для роботи з великими числами з підтримкою швидкого піднесення до степеня.

3. Створення навчальної системи RSA шифрування з візуалізацією всіх етапів.

4. Реалізація ефективних алгоритмів для обчислення чисел Фібоначчі та інших рекурентних послідовностей.

5. Розробка системи для підрахунку шляхів у графах з використанням матричного піднесення до степеня.

6. Створення криптографічної бібліотеки з constant-time реалізаціями модульної арифметики.

7. Дослідження та реалізація алгоритмів для дискретного логарифмування.

8. Розробка системи символьних обчислень з підтримкою швидкого піднесення до степеня поліномів.

9. Реалізація та оптимізація алгоритмів для обчислень в еліптичних кривих.

10. Створення бенчмарк наборів для тестування різних реалізацій на різних архітектурах процесорів.

Швидке перетворення Фур'є (FFT)

Теоретичні відомості

Швидке перетворення Фур'є (Fast Fourier Transform, FFT) є одним з найважливіших алгоритмів в обчислювальній математиці та цифровій обробці сигналів. Цей алгоритм дозволяє обчислити дискретне перетворення Фур'є (DFT) за час $O(n \log n)$ замість $O(n^2)$, що робить можливим аналіз сигналів та виконання багатьох інших обчислень у реальному часі.

Дискретне перетворення Фур'є

Дискретне перетворення Фур'є перетворює послідовність з N комплексних чисел $x_0, x_1, \ldots, x_{N-1}$ у послідовність комплексних чисел $X_0, X_1, \ldots, X_{N-1}$ за формулою:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n / N}$$

де: - X_k - k-та компонента перетворення Фур'є - x_n - n-тий елемент вхідної послідовності - i - уявна одиниця ($i^2 = -1$) - $e^{-2\pi i k n / N}$ - комплексний корінь з одиниці

Для зручності введемо позначення $\omega_N = e^{-2\pi i / N}$ - примітивний N-тий корінь з одиниці. Тоді формула набуває більш компактного вигляду:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot \omega_N^{kn}$$

Важливо розуміти, що ω_N - це комплексне число, яке при піднесенні до степеня N дає 1. Геометрично, це точка на одиничному колі в комплексній площині, яка ділить коло на N рівних частин.

Властивості коренів з одиниці

Примітивні корені з одиниці мають важливі властивості, які є ключовими для ефективності алгоритму FFT:

1. **Періодичність**: $\omega_N^{k+N} = \omega_N^k$

Це випливає з того, що $\omega_N^N = e^{-2\pi i} = 1$

2. **Симетрія**: $\omega_N^{k+N/2} = -\omega_N^k$ (для парного N)

Доведення: $\omega_N^{k+N/2} = \omega_N^k \cdot \omega_N^{N/2} = \omega_N^k \cdot e^{-\pi i} = -\omega_N^k$

3. **Властивість згортки**: $\omega_{2N}^{2k} = \omega_N^k$

Це означає, що корінь порядку 2N, піднесений до парного степеня, дає корінь порядку N

4. **Сума коренів**: $\sum_{k=0}^{N-1} \omega_N^k = 0$ (для N > 1)

Це випливає з формули суми геометричної прогресії.

Ідея алгоритму Кулі-Тьюкі

Алгоритм Кулі-Тьюкі базується на принципі "розділяй та володарюй". Основна ідея полягає в рекурсивному розбитті DFT розміру N на два DFT розміру N/2. Розглянемо випадок, коли N є степенем двійки: $N = 2^m$ для деякого натурального m.

Розділимо вхідну послідовність на елементи з парними та непарними індексами:

- Парні: $x_0, x_2, x_4, \ldots, x_{N-2}$

- Непарні: $x_1, x_3, x_5, \ldots, x_{N-1}$

Тоді DFT можна переписати як:

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} \cdot \omega_N^{2kn} + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot \omega_N^{k(2n+1)}$$

Винесемо спільний множник з другої суми:

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} \cdot \omega_N^{2kn} + \omega_N^k \sum_{n=0}^{N/2-1} x_{2n+1} \cdot \omega_N^{2kn}$$

Використовуючи властивість $\omega_N^{2kn} = \omega_{N/2}^{kn}$, отримуємо:

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} \cdot \omega_{N/2}^{kn} + \omega_N^k \cdot \sum_{n=0}^{N/2-1} x_{2n+1} \cdot \omega_{N/2}^{kn}$$

Позначимо:

- $E_k = \sum_{n=0}^{N/2-1} x_{2n} \cdot \omega_{N/2}^{kn}$ - DFT парних елементів

- $O_k = \sum_{n=0}^{N/2-1} x_{2n+1} \cdot \omega_{N/2}^{kn}$ - DFT непарних елементів

Тоді для $k = 0,1,\ldots, N/2-1$: - $X_k = E_k + \omega_N^k \cdot O_k$ - $X_{k+N/2} = E_k - \omega_N^k \cdot O_k$

Остання формула випливає з властивості симетрії: $\omega_N^{k+N/2} = -\omega_N^k$.

Рекурсивний алгоритм FFT має такий вигляд:

```
Функція FFT(x, N):
    // Базовий випадок
    якщо N == 1:
        повернути x

    // Перевірка, що N є степенем двійки
    якщо N не є степенем 2:
        помилка("N повинно бути степенем двійки")

    // Розділення на парні та непарні елементи
    парні = [x[0], x[2], x[4], ..., x[N-2]]
    непарні = [x[1], x[3], x[5], ..., x[N-1]]

    // Рекурсивні виклики
```

```
    E = FFT(парні, N/2)
    O = FFT(непарні, N/2)

    // Об'єднання результатів
    X = масив[N]
    для k від 0 до N/2 - 1:
        // ω_N^k = e^(-2πik/N)
        ω = cos(2πk/N) - i·sin(2πk/N)
        t = ω · O[k]
        X[k] = E[k] + t
        X[k + N/2] = E[k] - t

    повернути X
```

Ітеративний алгоритм FFT

Ітеративна версія FFT уникає накладних витрат на рекурсію та є більш ефективною на практиці. Вона використовує техніку бітової інверсії для переупорядкування елементів.

Бітова інверсія: Для числа в двійковому представленні, бітова інверсія означає зчитування бітів у зворотному порядку. Наприклад, для 3-бітових чисел:

- 000 → 000 (0 → 0)
- 001 → 100 (1 → 4)
- 010 → 010 (2 → 2)
- 011 → 110 (3 → 6)
- 100 → 001 (4 → 1)
- 101 → 101 (5 → 5)
- 110 → 011 (6 → 3)
- 111 → 111 (7 → 7)

```
Функція бітоваІнверсія(i, log2N):
    результат = 0
    для j від 0 до log2N - 1:
        якщо (i >> j) & 1:
            результат |= 1 << (log2N - 1 - j)
    повернути результат

Функція FFT_ітеративна(x, N):
    // Перевірка розміру
    log2N = log2(N)
```

```
    якщо 2^log2N != N:
        помилка("N повинно бути степенем двійки")

    // Копіювання з бітовою інверсією
    X = масив[N]
    для i від 0 до N - 1:
        j = бітоваІнверсія(i, log2N)
        X[j] = x[i]

    // Ітеративне обчислення
    для s від 1 до log2N:
        m = 2^s  // розмір поточних підзадач
        ωm = e^(-2πi/m)  // примітивний m-тий корінь

        для k від 0 до N - 1 кроком m:
            ω = 1
            для j від 0 до m/2 - 1:
                t = ω · X[k + j + m/2]
                u = X[k + j]
                X[k + j] = u + t
                X[k + j + m/2] = u - t
                ω = ω · ωm

    повернути X
```

Обернене FFT

Обернене перетворення Фур'є (IFFT) відновлює початкову послідовність з її спектру:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{2\pi i k n / N}$$

Зверніть увагу на три відмінності від прямого перетворення:

1. Знак в експоненті змінився з мінуса на плюс

2. З'явився нормалізуючий множник $\frac{1}{N}$

3. Сума йде по k, а не по n

Важливо відзначити, що IFFT можна реалізувати через той самий алгоритм FFT:

```
Функція IFFT(X, N):
    // Замінюємо ω_N на спряжене значення
    // Це еквівалентно зміні знаку в експоненті
    для i від 0 до N - 1:
```

```
        X[i] = комплексноСпряжене(X[i])

    // Виконуємо пряме FFT
    x = FFT(X, N)

    // Повертаємо спряження та нормалізуємо
    для і від 0 до N - 1:
        x[i] = комплексноСпряжене(x[i]) / N

    повернути x
```

FFT для дійсних чисел

Коли вхідні дані є дійсними числами, спектр має властивість ермітової симетрії:

$$X_{N-k} = \overline{X_k}$$

де $\overline{X_k}$ означає комплексне спряження. Це дозволяє зменшити кількість обчислень майже вдвічі:

```
Функція FFT_дійсне(x, N):
    // x - масив дійсних чисел
    // Використовуємо властивість: X[N-k] = conj(X[k])

    // Обчислюємо FFT для N/2 комплексних чисел
    z = масив[N/2]
    для і від 0 до N/2 - 1:
        z[i] = complex(x[2i], x[2i+1])

    Z = FFT(z, N/2)

    // Відновлюємо повний спектр
    X = масив[N]
    для k від 0 до N/2 - 1:
        // Розділяємо парну та непарну частини
        Ek = (Z[k] + conj(Z[(N/2-k) mod N/2])) / 2
        Ok = -i · (Z[k] - conj(Z[(N/2-k) mod N/2])) / 2

        ωk = e^(-2πi·k/N)
        X[k] = Ek + ωk · Ok
        X[k + N/2] = Ek - ωk · Ok

    повернути X
```

Розглянемо обчислення FFT для послідовності довжини N = 8:

x = [1, 2, 3, 4, 5, 6, 7, 8]

Крок 1: Бітова інверсія

Переупорядковуємо елементи згідно з бітовою інверсією індексів:

Індекс	Двійковий	Інвертований	Новий індекс	Значення
0	000	000	0	x[0] = 1
1	001	100	4	x[4] = 5
2	010	010	2	x[2] = 3
3	011	110	6	x[6] = 7
4	100	001	1	x[1] = 2
5	101	101	5	x[5] = 6
6	110	011	3	x[3] = 4
7	111	111	7	x[7] = 8

Переупорядкований масив: [1, 5, 3, 7, 2, 6, 4, 8]

Крок 2: Перший рівень (m = 2)

На цьому рівні об’єднуємо елементи попарно. Для m = 2, $\omega_2 = e^{-\pi i} = -1$:

- X[0] = 1 + 1·5 = 1 + 5 = 6
- X[1] = 1 - 1·5 = 1 - 5 = -4
- X[2] = 3 + 1·7 = 3 + 7 = 10
- X[3] = 3 - 1·7 = 3 - 7 = -4
- X[4] = 2 + 1·6 = 2 + 6 = 8
- X[5] = 2 - 1·6 = 2 - 6 = -4
- X[6] = 4 + 1·8 = 4 + 8 = 12
- X[7] = 4 - 1·8 = 4 - 8 = -4

Стан після першого рівня: [6, -4, 10, -4, 8, -4, 12, -4]

Крок 3: Другий рівень (m = 4)

Для m = 4, $\omega_4 = e^{-\pi i/2} = -i$:

Обробляємо першу групу (індекси 0-3):

- $\omega_4^0 = 1$:
- X[0] = 6 + 1·10 = 16
- X[2] = 6 - 1·10 = -4
- $\omega_4^1 = -i$:

- X[1] = -4 + (-i)·(-4) = -4 + 4i

- X[3] = -4 - (-i)·(-4) = -4 - 4i

Обробляємо другу групу (індекси 4-7):

- $\omega_4^0 = 1$:

- X[4] = 8 + 1·12 = 20

- X[6] = 8 - 1·12 = -4

- $\omega_4^1 = -i$:

- X[5] = -4 + (-i)·(-4) = -4 + 4i

- X[7] = -4 - (-i)·(-4) = -4 - 4i

Крок 4: Третій рівень (m = 8)

Для m = 8, обчислюємо потрібні корені:

- $\omega_8^0 = 1$

- $\omega_8^1 = e^{-\pi i/4} = \frac{1}{\sqrt{2}} - \frac{i}{\sqrt{2}} \approx 0.707 - 0.707i$

- $\omega_8^2 = e^{-\pi i/2} = -i$

- $\omega_8^3 = e^{-3\pi i/4} = -\frac{1}{\sqrt{2}} - \frac{i}{\sqrt{2}} \approx -0.707 - 0.707i$

Фінальне об’єднання:

- X[0] = 16 + 1·20 = 36

- X[1] = (-4+4i) + (0.707-0.707i)·(-4+4i) ≈ -4 + 9.657i

- X[2] = -4 + (-i)·(-4) = -4 + 4i

- X[3] = (-4-4i) + (-0.707-0.707i)·(-4-4i) ≈ -4 + 1.657i

- X[4] = 16 - 1·20 = -4

- X[5] = (-4+4i) - (0.707-0.707i)·(-4+4i) ≈ -4 - 1.657i

- X[6] = -4 - (-i)·(-4) = -4 - 4i

- X[7] = (-4-4i) - (-0.707-0.707i)·(-4-4i) ≈ -4 - 9.657i

Оптимізації алгоритму

1. Метод Radix-4 FFT:

Замість розбиття на 2 частини, розбиваємо на 4, що зменшує кількість множень. Основна ідея - використовувати той факт, що для коренів четвертого степеня маємо $\omega_4 = i$, тому множення на ω_4^k зводиться до перестановки та зміни знаків:

```
Функція FFT_Radix4(x, N):
    якщо N == 1:
        повернути x

    якщо N == 2:
        повернути [x[0] + x[1], x[0] - x[1]]

    // Розбиття на 4 частини
    x0 = [x[4k] для k від 0 до N/4-1]
    x1 = [x[4k+1] для k від 0 до N/4-1]
    x2 = [x[4k+2] для k від 0 до N/4-1]
    x3 = [x[4k+3] для k від 0 до N/4-1]

    // Рекурсивні виклики
    X0 = FFT_Radix4(x0, N/4)
    X1 = FFT_Radix4(x1, N/4)
    X2 = FFT_Radix4(x2, N/4)
    X3 = FFT_Radix4(x3, N/4)

    // Об'єднання
    X = масив[N]
    для k від 0 до N/4 - 1:
        ω1 = e^(-2πi·k/N)
        ω2 = ω1²
        ω3 = ω1³

        // Використовуємо матрицю 4×4 для об'єднання
        t0 = X0[k] + X1[k] + X2[k] + X3[k]
        t1 = X0[k] + i·X1[k] - X2[k] - i·X3[k]
        t2 = X0[k] - X1[k] + X2[k] - X3[k]
        t3 = X0[k] - i·X1[k] - X2[k] + i·X3[k]

        X[k] = t0
        X[k + N/4] = ω1·t1
        X[k + N/2] = ω2·t2
        X[k + 3N/4] = ω3·t3

    повернути X
```

2. Метод Bluestein для довільного N:

Цей метод дозволяє обчислити FFT для будь-якого N, перетворюючи DFT на згортку:

$$X_k = \sum_{n=0}^{N-1} x_n \, \omega_N^{kn} = \sum_{n=0}^{N-1} \left(x_n \omega_N^{n^2/2} \right) \cdot \omega_N^{-(k-n)^2/2} \cdot \omega_N^{k^2/2}$$

```
Функція FFT_Bluestein(x, N):
    // Перетворюємо DFT на згортку
    M = найближчийСтепіньДвійки(2N - 1)

    // Формуємо послідовності
    a = масив[M]
    b = масив[M]

    для n від 0 до N - 1:
        // Чирп-послідовність
        w = e^(-πi·n²/N)
        a[n] = x[n] · w
        b[n] = conj(w)

    // Дзеркальне відображення b для згортки
    для n від 1 до N - 1:
        b[M - n] = b[n]

    // Обчислюємо згортку через FFT
    A = FFT(a, M)
    B = FFT(b, M)

    для k від 0 до M - 1:
        A[k] = A[k] · B[k]

    c = IFFT(A, M)

    // Витягуємо результат
    X = масив[N]
    для k від 0 до N - 1:
        X[k] = c[k] · e^(-πi·k²/N)

    повернути X
```

Область застосування

Швидке перетворення Фур’є має надзвичайно широке коло застосувань у різних галузях науки та техніки:

- **Цифрова обробка сигналів**:

- Спектральний аналіз для визначення частотного складу сигналів

- Цифрова фільтрація (ФНЧ, ФВЧ, смугові, режекторні фільтри)

- Стиснення аудіо (MP3, AAC) та відео (MPEG, H.264)
- Розпізнавання мовлення та синтез мови
- Обробка біомедичних сигналів (ЕКГ, ЕЕГ, МРТ)
- **Телекомунікації**:
- OFDM (Orthogonal Frequency Division Multiplexing) модуляція в Wi-Fi, LTE, 5G
- Цифрове телебачення DVB
- Модеми та DSL технології
- Радіолокація та обробка радарних сигналів
- GPS та системи навігації
- **Наукові обчислення**:
- Розв’язання диференціальних рівнянь в частинних похідних
- Обчислювальна гідродинаміка (CFD)
- Моделювання фізичних процесів
- Квантово-механічні розрахунки
- Обробка сейсмічних даних
- **Обробка зображень**:
- Фільтрація та покращення зображень
- Видалення шуму та артефактів
- Стиснення зображень (JPEG використовує DCT, споріднене з FFT)
- Розпізнавання образів та комп’ютерний зір
- Медична візуалізація (КТ, МРТ)
- **Астрономія**:
- Обробка сигналів з радіотелескопів
- Аналіз гравітаційних хвиль (LIGO)
- Пошук екзопланет методом транзитів
- Спектральний аналіз зоряного світла
- Обробка зображень з космічних телескопів
- **Фінанси та економіка**:
- Аналіз часових рядів

- Виявлення циклів та трендів

- Прогнозування ринкових коливань

- Аналіз високочастотної торгівлі

- Оцінка ризиків

- **Музика та аудіо**:

- Цифрові еквалайзери та фільтри

- Спектральні аналізатори та візуалізатори

- Автотюн та корекція висоти тону

- Синтез звуку та семплінг

- Розпізнавання музики (Shazam)

Приклади застосування

Приклад 1: Швидке множення великих чисел

FFT дозволяє множити два n-значних числа за час O(n log n) замість $O(n^2)$. Основна ідея - представити числа як поліноми та використати теорему про згортку:

Нехай маємо два числа: - $A = a_{n-1} \dots a_1 a_0 = \sum_{i=0}^{n-1} a_i \cdot 10^i$ - $B = b_{n-1} \dots b_1 b_0 = \sum_{i=0}^{n-1} b_i \cdot 10^i$

Їх добуток можна представити як згортку коефіцієнтів:

$$C = A \cdot B = \sum_{k=0}^{2n-2} c_k \cdot 10^k$$

де $c_k = \sum_{i=0}^{k} a_i \cdot b_{k-i}$

```
Функція множенняЧерезFFT(A, B):
    // A та B - великі числа, представлені як масиви цифр
    n = max(довжина(A), довжина(B))
    N = найближчийСтепіньДвійки(2n)

    // Доповнюємо нулями до степеня двійки
    a = доповнитиНулями(A, N)
    b = доповнитиНулями(B, N)

    // FFT в комплексній області
    fa = FFT(a, N)
    fb = FFT(b, N)

    // Поточкове множення спектрів
```

```
    fc = масив[N]
    для і від 0 до N - 1:
        fc[i] = fa[i] · fb[i]

    // Обернене FFT
    c = IFFT(fc, N)

    // Обробка переносів (нормалізація)
    результат = []
    перенос = 0
    для і від 0 до N - 1:
        сума = round(Re(c[i])) + перенос
        результат[i] = сума % 10
        перенос = сума / 10

    // Додаємо залишковий перенос
    поки перенос > 0:
        результат.додати(перенос % 10)
        перенос = перенос / 10

    повернути результат
```

Приклад 2: Фільтрація сигналу

Видалення шуму з аудіосигналу за допомогою частотної фільтрації. Розглянемо реалізацію низькочастотного фільтра:

```
Функція фільтруватиСигнал(сигнал, частотаДискретизації, частотаЗрізу)
:
    N = довжина(сигнал)

    // Додаємо вікно для зменшення артефактів
    вікно = вікноХеммінга(N)
    для і від 0 до N - 1:
        сигнал[i] = сигнал[i] · вікно[i]

    // Перетворення в частотну область
    спектр = FFT(сигнал, N)

    // Створення частотної характеристики фільтра
    для k від 0 до N - 1:
        // Визначаємо частоту для k-го біна
        якщо k <= N/2:
            частота = k · частотаДискретизації / N
        інакше:
            частота = (k - N) · частотаДискретизації / N
```

```
        // Застосовуємо фільтр Баттерворта
        якщо abs(частота) <= частотаЗрізу:
            H = 1 / sqrt(1 + (частота/частотаЗрізу)^(2·порядок))
        інакше:
            H = 0

        спектр[k] = спектр[k] · H

    // Повернення в часову область
    відфільтрований = IFFT(спектр, N)

    // Повертаємо дійсну частину
    повернути Re(відфільтрований)

Функція вікноХеммінга(N):
    вікно = масив[N]
    для n від 0 до N - 1:
        вікно[n] = 0.54 - 0.46 · cos(2π·n/(N-1))
    повернути вікно
```

Приклад 3: Стиснення зображень

Використання 2D FFT для стиснення зображень з втратами:

```
Функція FFT2D(зображення, рядки, стовпці):
    // FFT по рядках
    проміжний = масив[рядки][стовпці]
    для і від 0 до рядки - 1:
        проміжний[і] = FFT(зображення[і], стовпці)

    // FFT по стовпцях
    результат = масив[рядки][стовпці]
    для j від 0 до стовпці - 1:
        стовпець = [проміжний[і][j] для і від 0 до рядки - 1]
        трансформований = FFT(стовпець, рядки)
        для і від 0 до рядки - 1:
            результат[і][j] = трансформований[і]

    повернути результат

Функція стиснутиЗображення(зображення, коефіцієнтСтиснення):
    рядки = висота(зображення)
    стовпці = ширина(зображення)

    // 2D FFT
    спектр = FFT2D(зображення, рядки, стовпці)
```

```
    // Обчислюємо енергію спектральних компонент
    енергії = []
    для і від 0 до рядки - 1:
        для j від 0 до стовпці - 1:
            енергія = |спектр[i][j]|²
            енергії.додати((енергія, i, j))

    // Сортуємо за енергією
    енергії.сортувати(заСпаданням)

    // Залишаємо тільки найбільш енергетичні компоненти
    кількістьКомпонент = round(рядки · стовпці / коефіцієнтСтиснення)

    стиснутийСпектр = масив[рядки][стовпці] // заповнений нулями
    для k від 0 до кількістьКомпонент - 1:
        (енергія, i, j) = енергії[k]
        стиснутийСпектр[i][j] = спектр[i][j]

    повернути стиснутийСпектр

Функція відновитиЗображення(стиснутийСпектр):
    повернути Re(IFFT2D(стиснутийСпектр))
```

Переваги та недоліки

Переваги

- Швидкість: Зменшення складності з $O(n^2)$ до $O(n\ log\ n)$ робить можливою обробку великих масивів даних у реальному часі. Для $N = 10^6$, це різниця між 10^{12} та $2{\times}10^7$ операцій - прискорення у 50000 разів!

- Точність: При правильній реалізації FFT дає точно такий же результат, як і пряме обчислення DFT, з точністю до помилок округлення.

- Універсальність: Може бути адаптований для різних типів даних (дійсні, комплексні) та розмірностей (1D, 2D, 3D FFT). Існують спеціалізовані версії для симетричних даних.

- Оборотність: IFFT дозволяє точно відновити початкові дані зі спектру без втрат інформації.

- Паралелізація: Структура алгоритму типу “метелик” дозволяє ефективну паралельну реалізацію на GPU та багатоядерних процесорах.

- Апаратна підтримка: Існують спеціалізовані DSP процесори та FPGA реалізації, оптимізовані для FFT. Бібліотеки як FFTW автоматично вибирають оптимальний алгоритм.

Недоліки

- Обмеження на розмір: Базовий алгоритм Кулі-Тьюкі працює тільки для N = 2^k. Для інших розмірів потрібні спеціальні алгоритми (Bluestein, Prime Factor FFT), які можуть бути повільнішими.

- Використання пам'яті: Потребує O(N) додаткової пам'яті для зберігання проміжних результатів та комплексних чисел (навіть для дійсних вхідних даних).

- Числові похибки: Накопичення помилок округлення при роботі з числами з плаваючою комою. Для N точок похибка ~O(log N × ε), де ε - машинна точність.

- Складність оптимальної реалізації: Досягнення максимальної продуктивності вимагає врахування кеш-пам'яті, векторизації, вибору оптимального алгоритму для конкретного N.

- Латентність: Для обробки в реальному часі потрібно накопичити весь блок з N відліків перед обробкою, що створює затримку N/Fs секунд.

- Спектральний витік: При обробці неперіодичних сигналів виникають артефакти через розривність на краях, що вимагає використання віконних функцій.

Контрольні запитання

1. У чому полягає основна ідея алгоритму швидкого перетворення Фур'є?

2. Які властивості коренів з одиниці використовуються в алгоритмі FFT?

3. Чому алгоритм Кулі-Тьюкі вимагає, щоб N було степенем двійки?

4. Як працює бітова інверсія в ітеративному алгоритмі FFT?

5. Яка різниця між прямим та оберненим перетворенням Фур'є?

6. Як можна оптимізувати FFT для дійсних чисел?

7. Що таке Radix-4 FFT і які його переваги?

8. Як алгоритм Bluestein дозволяє обчислити FFT для довільного N?

9. Чому FFT ефективний для множення великих чисел?

10. Які основні джерела числових похибок у FFT?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму швидкого перетворення Фур'є

2. Мета

Вивчити принципи роботи алгоритму FFT, реалізувати різні варіанти алгоритму, дослідити їх ефективність та застосувати для практичних задач обробки сигналів.

3. Завдання

- Реалізувати рекурсивну версію FFT алгоритму Кулі-Тьюкі
- Реалізувати ітеративну версію з бітовою інверсією
- Реалізувати обернене FFT
- Створити оптимізовану версію для дійсних чисел
- Реалізувати 2D FFT для обробки зображень
- Порівняти швидкість FFT з прямим обчисленням DFT
- Застосувати FFT для фільтрації аудіосигналу
- Реалізувати множення великих чисел через FFT

4. Опис

Лабораторна робота передбачає поетапну реалізацію різних версій алгоритму FFT, починаючи з базової рекурсивної версії до оптимізованих варіантів. Особлива увага приділяється правильній роботі з комплексними числами та дослідженню числової стійкості алгоритму.

5. Порядок виконання

- Реалізувати клас для роботи з комплексними числами
- Реалізувати обчислення коренів з одиниці
- Створити та протестувати рекурсивну версію FFT
- Реалізувати функцію бітової інверсії
- Створити ітеративну версію FFT
- Реалізувати IFFT через модифікацію FFT
- Оптимізувати для дійсних вхідних даних
- Реалізувати 2D FFT
- Створити демонстраційні програми
- Провести аналіз продуктивності

6. Результат

Бібліотека функцій для швидкого перетворення Фур'є з підтримкою різних типів даних та розмірностей. Демонстраційні програми, що показують застосування FFT для обробки сигналів, зображень та швидкого множення.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас комплексних чисел з усіма операціями
- Функції обчислення коренів з одиниці
- Рекурсивна реалізація FFT
- Ітеративна реалізація FFT
- Функція бітової інверсії
- Реалізація IFFT
- Оптимізований FFT для дійсних чисел
- 2D FFT
- Тестові функції

2. Продемонструвати роботу додатку:

- Порівняння результатів FFT з прямим DFT
- Графіки спектрів різних сигналів
- Фільтрація зашумленого сигналу
- Множення великих чисел
- Обробка зображення через 2D FFT
- Графіки порівняння швидкості

3. Бути готовим відповісти на питання:

- Чому FFT швидший за пряме обчислення DFT?
- Як працює об'єднання результатів у алгоритмі Кулі-Тьюкі?
- Що таке частота Найквіста і як вона пов'язана з FFT?
- Як вибрати розмір FFT для конкретного застосування?
- Які існують альтернативи алгоритму Кулі-Тьюкі?

Тести

1. Яка часова складність алгоритму FFT?

- 1) O(n)

- 2) O(n log n)
- 3) $O(n^2)$
- 4) $O(n^3)$

2. Для якого N працює базовий алгоритм Кулі-Тьюкі?
- 1) Будь-яке N
- 2) N = просте число
- 3) N = 2^k
- 4) N = парне число

3. Що таке ω_N в контексті FFT?
- 1) Частота дискретизації
- 2) Примітивний корінь з одиниці
- 3) Амплітуда сигналу
- 4) Фаза сигналу

4. Скільки комплексних множень потрібно для FFT розміру N?
- 1) N
- 2) N^2
- 3) $(N/2) \log_2 N$
- 4) $N \log_2 N$

5. Як обчислити IFFT через FFT?
- 1) Змінити знак уявної частини
- 2) Використати спряжені корені та нормалізувати
- 3) Виконати FFT двічі
- 4) Змінити порядок елементів

Відповіді:

1-2,

2-3,

3-2,

4-3,

5-2

Тести підвищеної складності

1. Яка мінімальна кількість дійсних множень для FFT розміру 8?

- 1) 2
- 2) 4
- 3) 8
- 4) 12

2. Що таке обертальний множник (**twiddle factor**) у контексті FFT?

- 1) Коефіцієнт масштабування
- 2) Корінь з одиниці ω_N^k
- 3) Індекс після бітової інверсії
- 4) Розмір підзадачі

3. Яка властивість спектру дійсного сигналу?

- 1) Спектр теж дійсний
- 2) Спектр суто уявний
- 3) X[k] = conj(X[N-k])
- 4) X[k] = -X[N-k]

4. Скільки операцій економить Radix-4 FFT порівняно з Radix-2?

- 1) 25% множень
- 2) 50% множень
- 3) 25% додавань
- 4) 50% додавань

5. Що таке zero-padding в контексті FFT?

- 1) Видалення нулів
- 2) Додавання нулів для інтерполяції спектру
- 3) Обнулення малих коефіцієнтів
- 4) Ініціалізація масиву

6. Яка точність FFT при використанні 64-бітних float?

- 1) Точний результат
- 2) Похибка ~10^-16
- 3) Похибка ~10^-8

- 4) Похибка ~10^-4

7. Що таке частота Найквіста для сигналу з частотою дискретизації Fs?

- 1) Fs

- 2) Fs/2

- 3) 2*Fs

- 4) Fs^2

8. Як називається ефект накладання спектрів?

- 1) Aliasing

- 2) Windowing

- 3) Leakage

- 4) Padding

9. Яка складність алгоритму Bluestein для довільного N?

- 1) O(N)

- 2) O(N log N)

- 3) $O(N^2)$

- 4) $O(N^3)$

10. Що таке in-place FFT?

- 1) FFT без рекурсії

- 2) FFT без додаткової пам’яті

- 3) FFT на місці вхідного масиву

- 4) FFT з фіксованою точкою

Відповіді:

1-1. Для FFT-8 потрібно лише 2 множення (решта - множення на ±1, ±i)

2-2. Twiddle factor - це корені з одиниці, що використовуються при об’єднанні

3-3. Спектр дійсного сигналу має ермітову симетрію

4-1. Radix-4 зменшує кількість множень приблизно на 25%

5-2. Zero-padding збільшує роздільну здатність спектру через інтерполяцію

6-2. Похибка визначається машинним епсилон ~2.22×10^-16

7-2. Частота Найквіста дорівнює половині частоти дискретизації

8-1. Aliasing - накладання спектрів при недостатній частоті дискретизації

9-2. Bluestein зводить DFT до згортки, яка обчислюється через FFT

10-3. In-place FFT використовує той самий масив для вхідних даних та результату

Завдання для самостійної роботи

1. Реалізуйте алгоритм FFT для випадку N = 3^k (Radix-3 FFT).

2. Дослідіть та реалізуйте алгоритм Prime Factor FFT для $N = p_1 \times p_2$, де p_1 та p_2 - взаємно прості.

3. Створіть програму для спектрального аналізу музичних файлів з візуалізацією спектрограми.

4. Реалізуйте алгоритм швидкої згортки двох сигналів через FFT.

5. Розробіть адаптивний фільтр на основі FFT для видалення періодичних завад.

6. Дослідіть вплив різних віконних функцій на спектральний аналіз.

7. Реалізуйте паралельну версію FFT з використанням OpenMP або CUDA.

8. Створіть програму для стиснення зображень на основі 2D FFT з різними рівнями компресії.

9. Реалізуйте алгоритм швидкого косинусного перетворення (FCT) через FFT.

10. Дослідіть числову стійкість FFT для різних порядків обходу (DIT vs DIF).

Теми лабораторних робіт

1. Реалізація та порівняння різних алгоритмів FFT (Кулі-Тьюкі, Radix-4, Split-Radix).

2. Розробка спектрального аналізатора аудіосигналів з графічним інтерфейсом.

3. Створення системи розпізнавання музичних нот на основі FFT.

4. Реалізація цифрових фільтрів (ФНЧ, ФВЧ, смуговий) через FFT.

5. Розробка програми для видалення шуму з зображень використовуючи 2D FFT.

6. Створення системи стиснення аудіо на принципах психоакустики та FFT.

7. Реалізація OFDM модулятора/демодулятора для передачі даних.

8. Розробка програми для аналізу вібрацій механічних систем через FFT.

9. Створення системи водяних знаків для зображень на основі FFT.

10. Реалізація алгоритмів обробки радарних сигналів з використанням FFT.

Алгоритм Штрассена для множення матриць

Теоретичні відомості

Алгоритм Штрассена, розроблений Фолькером Штрассеном у 1969 році, став революційним проривом у теорії обчислювальної складності. Цей алгоритм демонструє, що множення матриць можна виконати швидше, ніж за кубічний час, який вимагає стандартний алгоритм. Замість O(n^3) операцій, алгоритм Штрассена виконує множення за O(n^2.807) операцій, що дає значний виграш для великих матриць.

Стандартне множення матриць

Спочатку розглянемо класичний підхід до множення матриць. Для двох квадратних матриць A та B розміру $n \times n$, їх добуток $C = A \times B$ обчислюється за формулою:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

де C_{ij} - елемент матриці C у рядку i та стовпці j.

```
Функція стандартнеМноженняМатриць(A, B, n):
    C = нова_матриця[n][n]

    для і від 0 до n-1:
        для j від 0 до n-1:
            C[i][j] = 0
            для k від 0 до n-1:
                C[i][j] = C[i][j] + A[i][k] * B[k][j]

    повернути C
```

Цей алгоритм вимагає n^3 множень та n^3 - n^2 додавань, що дає загальну складність O(n^3).

Основна ідея алгоритму Штрассена

Ключова ідея Штрассена полягає в тому, щоб зменшити кількість рекурсивних множень з 8 до 7 при розбитті матриць на блоки. Хоча це збільшує кількість додавань, асимптотично алгоритм стає швидшим.

Розглянемо множення двох матриць 2×2:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Стандартний підхід вимагає 8 множень:

- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$

Формули Штрассена

Штрассен запропонував обчислювати сім проміжних матриць:

- $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
- $M_2 = (A_{21} + A_{22})B_{11}$
- $M_3 = A_{11}(B_{12} - B_{22})$
- $M_4 = A_{22}(B_{21} - B_{11})$
- $M_5 = (A_{11} + A_{12})B_{22}$
- $M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
- $M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$

Тоді елементи результуючої матриці обчислюються як:

- $C_{11} = M_1 + M_4 - M_5 + M_7$
- $C_{12} = M_3 + M_5$
- $C_{21} = M_2 + M_4$
- $C_{22} = M_1 - M_2 + M_3 + M_6$

Важливо переконатися, що ці формули дають правильний результат. Розглянемо, наприклад, обчислення C_{11}:

$$C_{11} = M_1 + M_4 - M_5 + M_7$$

Підставимо значення:

$$C_{11} = (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22}(B_{21} - B_{11}) - (A_{11} + A_{12})B_{22} + (A_{12} - A_{22})(B_{21} + B_{22})$$

Розкриємо дужки та спростимо вираз, щоб отримати

$C_{11} = A_{11}B_{11} + A_{12}B_{21}$,

що відповідає стандартній формулі.

Рекурсивний алгоритм Штрассена має такий вигляд:

```
Функція множенняШтрассена(A, B, n):
    // Базовий випадок
    якщо n == 1:
        повернути [[A[0][0] * B[0][0]]]

    // Якщо n невелике, використати стандартний алгоритм
    якщо n <= ПОРІГ:
        повернути стандартнеМноженняМатриць(A, B, n)

    // Доповнення до степеня двійки (якщо потрібно)
    якщо n не є степенем 2:
        новийРозмір = найближчийСтепіньДвійки(n)
        A = доповнитиНулями(A, новийРозмір)
        B = доповнитиНулями(B, новийРозмір)
        n = новийРозмір

    // Розбиття матриць на блоки
    половина = n / 2

    A11, A12, A21, A22 = розбитиНаБлоки(A, половина)
    B11, B12, B21, B22 = розбитиНаБлоки(B, половина)

    // Обчислення проміжних матриць
    M1 = множенняШтрассена(додати(A11, A22), додати(B11, B22), полови
на)
    M2 = множенняШтрассена(додати(A21, A22), B11, половина)
    M3 = множенняШтрассена(A11, відняти(B12, B22), половина)
    M4 = множенняШтрассена(A22, відняти(B21, B11), половина)
    M5 = множенняШтрассена(додати(A11, A12), B22, половина)
    M6 = множенняШтрассена(відняти(A21, A11), додати(B11, B12), полов
ина)
    M7 = множенняШтрассена(відняти(A12, A22), додати(B21, B22), полов
ина)

    // Обчислення блоків результуючої матриці
    C11 = додати(відняти(додати(M1, M4), M5), M7)
    C12 = додати(M3, M5)
    C21 = додати(M2, M4)
    C22 = додати(відняти(додати(M1, M3), M2), M6)

    // Збирання результату
    C = об'єднатиБлоки(C11, C12, C21, C22)

    // Видалення доповнення (якщо було)
    якщо початковийРозмір != n:
```

```
        C = обрізати(C, початковийРозмір)

    повернути C
```

Допоміжні функції

```
Функція розбитиНаБлоки(матриця, половина):
    n = розмір(матриця)

    блок11 = нова_матриця[половина][половина]
    блок12 = нова_матриця[половина][половина]
    блок21 = нова_матриця[половина][половина]
    блок22 = нова_матриця[половина][половина]

    для i від 0 до половина-1:
        для j від 0 до половина-1:
            блок11[i][j] = матриця[i][j]
            блок12[i][j] = матриця[i][j + половина]
            блок21[i][j] = матриця[i + половина][j]
            блок22[i][j] = матриця[i + половина][j + половина]

    повернути блок11, блок12, блок21, блок22

Функція додати(A, B):
    n = розмір(A)
    C = нова_матриця[n][n]

    для i від 0 до n-1:
        для j від 0 до n-1:
            C[i][j] = A[i][j] + B[i][j]

    повернути C

Функція відняти(A, B):
    n = розмір(A)
    C = нова_матриця[n][n]

    для i від 0 до n-1:
        для j від 0 до n-1:
            C[i][j] = A[i][j] - B[i][j]

    повернути C
```

Аналіз складності

Нехай T(n) - час виконання алгоритму для матриць розміру n×n. Тоді рекурентне співвідношення має вигляд:

$$T(n) = 7T(n/2) + O(n^2)$$

де: - 7T(n/2) - час для семи рекурсивних множень матриць розміру n/2×n/2 - O(n^2) - час для додавань та віднімань матриць

Використовуючи основну теорему про рекурентні співвідношення:

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.807})$$

Розглянемо множення двох матриць 4×4:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, \quad B = \begin{bmatrix} 16 & 15 & 14 & 13 \\ 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

Крок 1: Розбиття на блоки

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}, A_{12} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix}, A_{22} = \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 16 & 15 \\ 12 & 11 \end{bmatrix}, B_{12} = \begin{bmatrix} 14 & 13 \\ 10 & 9 \end{bmatrix}$$

$$B_{21} = \begin{bmatrix} 8 & 7 \\ 4 & 3 \end{bmatrix}, B_{22} = \begin{bmatrix} 6 & 5 \\ 2 & 1 \end{bmatrix}$$

Крок 2: Обчислення допоміжних матриць для M_1

Для $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$:

$$A_{11} + A_{22} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix} = \begin{bmatrix} 12 & 14 \\ 20 & 22 \end{bmatrix}$$

$$B_{11} + B_{22} = \begin{bmatrix} 16 & 15 \\ 12 & 11 \end{bmatrix} + \begin{bmatrix} 6 & 5 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 22 & 20 \\ 14 & 12 \end{bmatrix}$$

Тепер множимо ці матриці 2×2 (рекурсивно або напряму для базового випадку):

$$M_1 = \begin{bmatrix} 12 & 14 \\ 20 & 22 \end{bmatrix} \times \begin{bmatrix} 22 & 20 \\ 14 & 12 \end{bmatrix} = \begin{bmatrix} 460 & 408 \\ 748 & 664 \end{bmatrix}$$

Крок 3: Аналогічно обчислюємо M_2 - M_7

Процес обчислення інших матриць M аналогічний. Кожна вимагає додавання/віднімання відповідних блоків та їх множення.

Крок 4: Збирання результату

Після обчислення всіх M матриць, обчислюємо блоки результату: - $C_{11} = M_1 + M_4 - M_5 + M_7$ - $C_{12} = M_3 + M_5$ - $C_{21} = M_2 + M_4$ - $C_{22} = M_1 - M_2 + M_3 + M_6$

Фінальний результат:

$$C = \begin{bmatrix} 80 & 70 & 60 & 50 \\ 240 & 214 & 188 & 162 \\ 400 & 358 & 316 & 274 \\ 560 & 502 & 444 & 386 \end{bmatrix}$$

Оптимізації алгоритму

1. Вибір порогу переключення: На практиці для малих матриць накладні витрати алгоритму Штрассена перевищують виграш від зменшення кількості множень. Тому важливо вибрати оптимальний поріг (зазвичай між 32 та 128), нижче якого використовується стандартний алгоритм.

```
Функція адаптивнеМноженняШтрассена(A, B, n):
    ОПТИМАЛЬНИЙ_ПОРІГ = 64

    якщо n <= ОПТИМАЛЬНИЙ_ПОРІГ:
        повернути стандартнеМноженняМатриць(A, B, n)
    інакше:
        повернути множенняШтрассена(A, B, n)
```

2. Уникнення копіювання даних: Замість створення нових матриць для блоків, можна використовувати індекси для роботи з підматрицями:

```
Функція множенняШтрассенаЗІндексами(A, B, n, rowA, colA, rowB, colB):
    // Працює з підматрицями без копіювання
    // rowA, colA - початкові індекси для матриці A
    // rowB, colB - початкові індекси для матриці B

    якщо n <= ПОРІГ:
        повернути множенняБлоків(A, B, n, rowA, colA, rowB, colB)

    половина = n / 2

    // Рекурсивні виклики з відповідними зміщеннями індексів
    // ...
```

3. Паралелізація: Сім рекурсивних множень можна виконувати паралельно:

```
Функція паралельнийШтрассен(A, B, n):
    якщо n <= ПОРІГ_ПАРАЛЕЛІЗМУ:
        повернути множенняШтрассена(A, B, n)

    // Обчислення блоків
    половина = n / 2
    A11, A12, A21, A22 = розбитиНаБлоки(A, половина)
    B11, B12, B21, B22 = розбитиНаБлоки(B, половина)

    // Паралельне обчислення M матриць
```

```
    паралельно {
        M1 = множенняШтрассена(додати(A11, A22), додати(B11, B22), по
ловина)
        M2 = множенняШтрассена(додати(A21, A22), B11, половина)
        M3 = множенняШтрассена(A11, відняти(B12, B22), половина)
        M4 = множенняШтрассена(A22, відняти(B21, B11), половина)
        M5 = множенняШтрассена(додати(A11, A12), B22, половина)
        M6 = множенняШтрассена(відняти(A21, A11), додати(B11, B12), п
оловина)
        M7 = множенняШтрассена(відняти(A12, A22), додати(B21, B22), п
оловина)
    }

    // Збирання результату
    повернути об'єднатиБлоки(обчислитиC11(M1,M4,M5,M7), ...)
```

Узагальнення та варіації

Алгоритм Віноградa: Модифікація алгоритму Штрассена, яка зменшує кількість додавань з 18 до 15 для кожного рівня рекурсії.

Алгоритми вищого порядку: Існують алгоритми з кращою асимптотикою, наприклад: - Алгоритм Копперсміта-Винограда: O(n^2.376) - Найкращий відомий результат (2020): O(n^2.3728596)

Однак ці алгоритми мають настільки великі константи, що стають практичними лише для астрономічно великих матриць.

Область застосування

Алгоритм Штрассена знаходить застосування в багатьох областях, де потрібне ефективне множення великих матриць:

- Наукові обчислення: Моделювання фізичних процесів, розв’язання систем лінійних рівнянь, обчислювальна гідродинаміка, квантова механіка.

- Комп’ютерна графіка: Трансформації в 3D графіці, рендеринг, обробка зображень великої роздільної здатності.

- Машинне навчання: Навчання нейронних мереж, обчислення в глибоких мережах, матричні операції в алгоритмах оптимізації.

- Обробка сигналів: Цифрова обробка сигналів, стиснення даних, криптографія.

- Економетрика та статистика: Регресійний аналіз великих наборів даних, обчислення коваріаційних матриць.

- Біоінформатика: Аналіз геномних даних, моделювання білкових структур.

Приклади застосування

Приклад 1: Обчислення степенів матриці

Для обчислення A^k, де k - велике число, можна використати швидке піднесення до степеня з алгоритмом Штрассена:

```
Функція матрицяВСтепені(A, k):
    якщо k == 0:
        повернути одиничнуМатрицю(розмір(A))
    якщо k == 1:
        повернути A

    якщо k парне:
        половина = матрицяВСтепені(A, k/2)
        повернути множенняШтрассена(половина, половина, розмір(A))
    інакше:
        повернути множенняШтрассена(A, матрицяВСтепені(A, k-1), розмі
р(A))
```

Приклад 2: Розв’язання систем лінійних рівнянь

При використанні ітераційних методів часто потрібно множити великі матриці. Наприклад, в методі спряжених градієнтів:

```
Функція спряженіГрадієнти(A, b, x0):
    x = x0
    r = b - множенняШтрассена(A, x, n)
    p = r

    для k від 0 до максІтерацій:
        Ap = множенняШтрассена(A, p, n)
        alpha = (r · r) / (p · Ap)
        x = x + alpha * p
        r_new = r - alpha * Ap

        якщо норма(r_new) < epsilon:
            вийти

        beta = (r_new · r_new) / (r · r)
        p = r_new + beta * p
        r = r_new

    повернути x
```

Приклад 3: Обробка зображень

При застосуванні фільтрів до великих зображень часто використовується матричне множення:

```
Функція застосуватиФільтр(зображення, фільтр):
    // Перетворення зображення в матрицю
    матрицяЗображення = зображенняВМатрицю(зображення)

    // Застосування фільтру через матричне множення
    результат = множенняШтрассена(фільтр, матрицяЗображення, розмір)

    // Перетворення назад у зображення
    повернути матрицюВЗображення(результат)
```

Переваги та недоліки

Переваги

- Асимптотична ефективність: Для великих матриць забезпечує значну економію часу порівняно зі стандартним алгоритмом. Виграш стає помітним для матриць розміром більше 512×512.

- Теоретична важливість: Перший алгоритм, що показав можливість множення матриць швидше ніж за $O(n^3)$, відкривши нову область досліджень.

- Можливість паралелізації: Сім незалежних рекурсивних викликів можна виконувати паралельно.

- Застосовність до різних типів даних: Працює для будь-якого кільця, не тільки для чисел.

Недоліки

- Великі константи: Накладні витрати на рекурсію та додаткові операції додавання/віднімання роблять алгоритм неефективним для малих матриць.

- Числова нестабільність: Додаткові операції додавання та віднімання можуть накопичувати похибки округлення.

- Складність реалізації: Правильна та ефективна реалізація вимагає уважного програмування та оптимізації.

- Використання пам’яті: Потребує додаткової пам’яті для проміжних результатів, що може бути критичним для дуже великих матриць.

- Неоптимальність для розріджених матриць: Для матриць з багатьма нулями існують спеціалізовані алгоритми, які працюють краще.

Контрольні запитання

1. У чому полягає основна ідея алгоритму Штрассена?

2. Скільки множень виконує алгоритм Штрассена для матриць 2×2?

3. Яка асимптотична складність алгоритму Штрассена?

4. Чому алгоритм Штрассена неефективний для малих матриць?

5. Як обчислюється матриця M_1 в алгоритмі Штрассена?

6. Які переваги дає використання порогу переключення?

7. Як можна паралелізувати алгоритм Штрассена?

8. Чи можна застосувати алгоритм Штрассена для прямокутних матриць?

9. Які існують узагальнення алгоритму Штрассена?

10. В яких випадках краще використовувати стандартний алгоритм множення?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму Штрассена для множення матриць

2. Мета

Вивчити принципи роботи алгоритму Штрассена, реалізувати алгоритм з оптимізаціями, дослідити його ефективність порівняно зі стандартним алгоритмом множення матриць.

3. Завдання

- Реалізувати стандартний алгоритм множення матриць
- Реалізувати базовий алгоритм Штрассена
- Додати адаптивний поріг переключення
- Реалізувати версію без копіювання підматриць
- Дослідити вплив розміру матриць на продуктивність
- Порівняти точність обчислень
- Реалізувати паралельну версію алгоритму
- Застосувати для обчислення степенів матриці

4. Опис

Лабораторна робота передбачає поетапну реалізацію алгоритму Штрассена, починаючи з базової версії до оптимізованих варіантів. Особлива увага приділяється дослідженню практичної ефективності та вибору оптимальних параметрів.

5. Порядок виконання

- Реалізувати допоміжні функції для операцій з матрицями
- Реалізувати стандартне множення для порівняння
- Створити базову рекурсивну версію алгоритму Штрассена
- Додати обробку матриць довільного розміру
- Реалізувати адаптивний вибір порогу
- Оптимізувати роботу з пам’яттю
- Додати підтримку паралельного виконання
- Створити тестовий набір різних розмірів
- Виміряти та проаналізувати продуктивність
- Дослідити накопичення похибок

6. Результат

Ефективна реалізація алгоритму Штрассена з автоматичним вибором оптимальних параметрів. Звіт з аналізом продуктивності та рекомендаціями щодо використання алгоритму для різних розмірів матриць.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функції для базових операцій з матрицями
- Стандартний алгоритм множення
- Рекурсивна реалізація алгоритму Штрассена
- Функція адаптивного вибору алгоритму
- Оптимізована версія без копіювання
- Паралельна реалізація
- Тестові функції та вимірювання часу

2. Продемонструвати роботу додатку:

- Коректність роботи на тестових прикладах
- Графіки залежності часу від розміру матриць
- Порівняння зі стандартним алгоритмом
- Вплив порогу на продуктивність

- Ефективність паралелізації
- Аналіз точності обчислень

3. Бути готовим відповісти на питання:
- Чому алгоритм використовує саме 7 множень?
- Як довести коректність формул Штрассена?
- Який оптимальний поріг для вашої системи?
- Як масштабується паралельна версія?
- Які фактори впливають на вибір алгоритму?

Тести

1. Яка асимптотична складність алгоритму Штрассена?
- 1) $O(n^2)$
- 2) O(n^2.807)
- 3) $O(n^3)$
- 4) O(n log n)

2. Скільки рекурсивних множень виконує алгоритм Штрассена?
- 1) 4
- 2) 6
- 3) 7
- 4) 8

3. Для яких матриць алгоритм Штрассена найефективніший?
- 1) Малих ($n < 32$)
- 2) Середніх ($32 \leqslant n \leqslant 512$)
- 3) Великих ($n > 512$)
- 4) Розріджених

4. Що є основним недоліком алгоритму Штрассена?
- 1) Неправильні результати
- 2) Великі константи для малих матриць
- 3) Неможливість паралелізації
- 4) Працює тільки для квадратних матриць

5. Яке рекурентне співвідношення описує час роботи алгоритму?

- 1) $T(n) = 8T(n/2) + O(n^2)$

- 2) $T(n) = 7T(n/2) + O(n^2)$

- 3) $T(n) = 4T(n/2) + O(n)$

- 4) $T(n) = 2T(n/2) + O(n^2)$

Відповіді:

1-2,

2-3,

3-3,

4-2,

5-2

Тести підвищеної складності

1. Який алгоритм має найкращу відому асимптотику для множення матриць?

- 1) Штрассена O(n^2.807)

- 2) Копперсміта-Винограда O(n^2.376)

- 3) Вільямса O(n^2.3728596)

- 4) Карацуби O(n^1.585)

2. Скільки додавань/віднімань матриць виконує алгоритм Штрассена на одному рівні рекурсії?

- 1) 7

- 2) 14

- 3) 18

- 4) 22

3. Для якого типу матриць алгоритм Штрассена може давати найбільші числові похибки?

- 1) Одиничних

- 2) Діагональних

- 3) З елементами різних порядків

- 4) Ортогональних

4. Яка модифікація зменшує кількість додавань в алгоритмі Штрассена?

- 1) Алгоритм Карацуби

- 2) Алгоритм Винограда

- 3) Алгоритм Бінета

- 4) Алгоритм Паніна

5. Скільки тимчасової пам’яті потрібно для алгоритму Штрассена?

- 1) O(1)

- 2) O(n)

- 3) $O(n^2)$

- 4) O(log n)

6. Який поріг переключення зазвичай оптимальний на сучасних процесорах?

- 1) n = 16

- 2) n = 32-128

- 3) n = 256-512

- 4) n = 1024

7. Чому алгоритм Копперсміта-Винограда не використовується на практиці?

- 1) Патентні обмеження

- 2) Величезні константи

- 3) Числова нестабільність

- 4) Складність паралелізації

8. Скільки рівнів рекурсії має алгоритм Штрассена для матриць n×n?

- 1) n

- 2) $\log_2 n$

- 3) $\sqrt{n}$

- 4) n^2

9. Для яких полів працює алгоритм Штрассена?

- 1) Тільки дійсні числа

- 2) Тільки комплексні числа

- 3) Будь-яке кільце

- 4) Тільки цілі числа

10. Яка складність найкращого алгоритму множення для розріджених матриць?

- 1) $O(n^2)$

- 2) O(nnz)

- 3) O(n + nnz)

- 4) Залежить від структури

Відповіді:

1-3. Алгоритм Вільямса (2020) має найкращу відому асимптотику O(n^2.3728596)

2-3. Алгоритм виконує 18 додавань/віднімань матриць розміру n/2

3-3. Матриці з елементами різних порядків можуть призводити до втрати точності

4-2. Алгоритм Винограда зменшує кількість додавань з 18 до 15

5-3. Потрібно $O(n^2)$ пам'яті для зберігання проміжних матриць

6-2. Оптимальний поріг зазвичай між 32 та 128 залежно від архітектури

7-2. Константи настільки великі, що алгоритм стає практичним лише для астрономічних n

8-2. Глибина рекурсії дорівнює $\log_2 n$

9-3. Алгоритм працює для будь-якого кільця, не тільки чисел

10-4. Складність залежить від структури розрідженості та використаного алгоритму

Завдання для самостійної роботи

1. Реалізуйте алгоритм Винограда-Штрассена зі зменшеною кількістю додавань.

2. Дослідіть вплив різних стратегій обробки неквадратних матриць на продуктивність.

3. Реалізуйте гібридний алгоритм, що автоматично вибирає між Штрассеном та іншими методами.

4. Створіть версію алгоритму для множення комплексних матриць з оптимізаціями.

5. Розробіть методи зменшення числових похибок в алгоритмі Штрассена.

6. Реалізуйте алгоритм для обчислення добутку ланцюга матриць з використанням Штрассена.

7. Дослідіть ефективність алгоритму на різних архітектурах процесорів.

8. Створіть адаптивну паралельну версію з динамічним розподілом навантаження.

9. Реалізуйте алгоритм Штрассена для блочних матриць великого розміру.

10. Порівняйте різні стратегії вибору порогу переключення для різних типів даних.

Теми лабораторних робіт

1. Реалізація та оптимізація базового алгоритму Штрассена.

2. Дослідження числової стабільності алгоритму для різних типів матриць.

3. Розробка адаптивного алгоритму з автоматичним вибором методу множення.

4. Паралельна реалізація алгоритму Штрассена на багатоядерних системах.

5. Застосування алгоритму для розв’язання систем лінійних рівнянь.

6. Реалізація алгоритму для GPU з використанням CUDA/OpenCL.

7. Розробка бібліотеки матричних операцій з підтримкою алгоритму Штрассена.

8. Дослідження ефективності для спеціальних типів матриць.

9. Створення візуалізації роботи рекурсивного алгоритму.

10. Оптимізація алгоритму для конкретної апаратної платформи.

Частина VI: Просунуті та сучасні алгоритми

Частина VI цього підручника присвячена трьом фундаментальним напрямкам алгоритмічної науки, які визначають технологічне майбутнє: паралельним і розподіленим системам, алгоритмам для великих даних та потоків, а також алгоритмам машинного навчання. Ці теми не просто доповнюють одна одну — вони утворюють синергетичну тріаду, де кожен компонент підсилює можливості інших.

Сучасні обчислювальні виклики вимагають комплексного підходу. Коли Netflix щороку економить мільярд доларів завдяки рекомендаційним алгоритмам, або коли Google обробляє мільярди пошукових запитів за лічені мілісекунди, за цим стоїть злагоджена робота всіх трьох алгоритмічних парадигм. Паралельні системи забезпечують необхідну обчислювальну потужність, алгоритми для великих даних дозволяють ефективно працювати з колосальними обсягами інформації, а машинне навчання перетворює ці дані на інтелектуальні рішення.

Розглянемо реальний приклад: система автоматичного керування автомобілем Tesla. Вона використовує **паралельні алгоритми** для одночасної обробки даних з десятків сенсорів, застосовує **потокові алгоритми** для аналізу відеоданих у реальному часі зі швидкістю 36 ГБ на годину, і покладається на **алгоритми машинного навчання** для розпізнавання об’єктів та прийняття рішень. Без глибокого розуміння всіх трьох компонентів створення такої системи було б неможливим.

Розділ 13: Паралельні та розподілені системи — основа сучасних обчислень

У світі, де найпотужніший суперкомп’ютер El Capitan виконує 2 квінтильйони операцій за секунду, а квантовий чип Google Willow розв’язує за 5 хвилин задачі, на які класичним комп’ютерам знадобилося б більше часу, ніж існує Всесвіт, розуміння паралельних алгоритмів стає критично важливим. У цьому розділі ви опануєте:

Паралельні алгоритми — від базових принципів розпаралелювання до складних схем розподілу навантаження. Ви навчитеся думати паралельно, розбиваючи задачі на незалежні підзадачі та ефективно координуючи їх виконання. Дізнаєтеся, як Amazon AWS Parallel Computing Service масштабує обчислення від нуля до тисяч процесорів за лічені хвилини.

Lock-free структури даних — революційний підхід до синхронізації в багатопотокових системах. Ці алгоритми дозволяють тисячам потоків працювати одночасно без блокувань, забезпечуючи неймовірну продуктивність сучасних процесорів з десятками ядер.

Розподілені алгоритми — основа хмарних обчислень та інтернет-сервісів. Ви зрозумієте, як Google обробляє 8,5 мільярда пошукових запитів щодня, розподіляючи навантаження між мільйонами серверів, і як забезпечується узгодженість даних у глобальному масштабі.

MapReduce та його еволюція — від класичної парадигми до сучасних фреймворків Apache Spark та Flink. Навчитеся обробляти терабайти даних, використовуючи прості функціональні примітиви map і reduce.

Розділ 14: Алгоритми для великих даних

Кожну хвилину у світі завантажується 720 000 годин відео на YouTube, відправляється 231 мільйон електронних листів і публікується 490 000 відео в TikTok. Традиційні алгоритми безсилі перед таким потоком даних. У цьому розділі ви освоїте інструменти для роботи з Big Data:

Sketch-структури — геніальні ймовірнісні структури даних, які дозволяють аналізувати терабайтні потоки, використовуючи лише кілобайти пам'яті. Count-Min Sketch, HyperLogLog, Bloom Filter — ці алгоритми стали основою систем реального часу від Twitter до Netflix.

Алгоритми для потоків — мистецтво обробки даних, які ви бачите лише один раз. Дізнайтеся, як фінансові системи виявляють шахрайство серед мільйонів транзакцій у реальному часі, обробляючи кожну транзакцію за мікросекунди.

Структури для зовнішньої пам'яті — коли дані не поміщаються в оперативну пам'ять, потрібні спеціальні алгоритми. B-дерева, LSM-дерева та інші структури забезпечують ефективну роботу з петабайтними сховищами даних.

Cache-oblivious алгоритми — елегантне рішення для оптимізації роботи з ієрархією пам'яті без знання її параметрів. Ці алгоритми автоматично адаптуються до будь-якої системи, від смартфона до суперкомп'ютера.

Розділ 15: Алгоритми машинного навчання — штучний інтелект на практиці

Ринок штучного інтелекту досяг 196 мільярдів доларів у 2024 році, а 83% компаній вважають ШІ своїм стратегічним пріоритетом. За цими цифрами стоять конкретні алгоритми, які ви вивчите в цьому розділі:

Алгоритми регресії — від простої лінійної регресії до складних нелінійних моделей. Навчитеся передбачати ціни на нерухомість, прогнозувати продажі та оптимізувати бізнес-процеси.

Кластеризація — виявлення прихованих патернів у даних. K-means, DBSCAN, ієрархічна кластеризація — алгоритми, які допомагають сегментувати клієнтів, групувати документи та аналізувати соціальні мережі.

Класифікація — навчання комп'ютера розпізнавати категорії. Від дерев рішень до методів опорних векторів — освоїте алгоритми, які лежать в основі систем розпізнавання облич, діагностики захворювань та фільтрації спаму.

Нейронні мережі — революція глибокого навчання. Зрозумієте, як працюють трансформери, що стоять за ChatGPT, як згорткові мережі розпізнають зображення краще за людину, і як рекурентні мережі обробляють природну мову.

Зниження розмірності — мистецтво спрощення без втрати суті. PCA, t-SNE, автокодувальники — методи, які дозволяють візуалізувати багатовимірні дані та прискорювати навчання моделей у тисячі разів.

У цій Частині підручника, ми закладемо фундамент високоефективним обчисленням — технологіям, без яких неможливий жоден сучасний сервіс чи додаток.

Розділ 13: Паралельні та розподілені системи

Протягом десятиліть комп'ютерна індустрія жила за законом Мура: кількість транзисторів на чіпі подвоювалася кожні два роки, а разом з нею росла й швидкість процесорів. Програмісти могли просто чекати, і їхні програми автоматично ставали швидшими на новому обладнанні. Але близько 2005 року сталося щось надзвичайне — фізичні закони поставили жорсткий ліміт на швидкість окремого процесора. Тактові частоти застигли на позначці 3-4 ГГц, і подальше збільшення призводило до неприйнятного енергоспоживання та перегріву.

Індустрія знайшла елегантне рішення: якщо не можна зробити один процесор швидшим, давайте поставимо поруч багато процесорів. Так народилася ера багатоядерності. Спочатку два ядра, потім чотири, вісім… Сучасні процесори AMD Threadripper мають до 96 ядер, а графічні процесори NVIDIA H100 містять десятки тисяч обчислювальних елементів.

Парадокс паралелізму

Здавалося б, якщо у нас є N процесорів, то будь-яка задача має розв'язуватися в N разів швидше. Але Джин Амдал у 1967 році сформулював закон, який розвіяв ці ілюзії. Закон Амдала стверджує: якщо частка програми P може бути розпаралелена, а частка (1-P) залишається послідовною, то максимальне прискорення на N процесорах становить:

$$\text{Прискорення} = \frac{1}{(1-P) + \frac{P}{N}}$$

Це означає, що навіть якщо 90% програми можна розпаралелити, максимальне прискорення на нескінченній кількості процесорів становитиме лише 10 разів! Послідовна частина стає вузьким місцем, яке неможливо подолати додаванням процесорів.

Але людська винахідливість знайшла способи обійти це обмеження. Джон Густафсон запропонував інший погляд: замість прискорення фіксованої задачі, ми можемо розв'язувати більші задачі за той самий час. Це відкрило шлях до обчислень планетарного масштабу.

У цьому розділі ми досліджуємо чотири взаємопов'язані, але різні світи паралельних обчислень:

Паралельні алгоритми — мистецтво розділяй і володарюй

У підрозділі 13.1 ви познайомитеся з елегантними паралельними алгоритмами, які перевертають традиційне мислення. Bitonic sort — алгоритм сортування, який здається неефективним з послідовної точки зору ($O(n \log^2 n)$ операцій), але на паралельній архітектурі працює за час $O(\log^2 n)$ з n процесорами. Це як організувати тисячу людей для сортування карток: кожен знає лише про своїх сусідів, але разом вони створюють ідеальний порядок.

Підрозділ 13.2 відкриває двері в найскладнішу область паралельного програмування. Традиційна синхронізація через блокування (locks) створює вузькі місця: поки один потік тримає блокування, тисячі інших

чекають. Lock-free алгоритми вирішують цю проблему радикально — вони взагалі не використовують блокування.

У підрозділі 13.3 масштаб зростає від ядер процесора до планетарних мереж. Розподілені системи — це вже не просто паралельні обчислення, це координація незалежних комп'ютерів, розкиданих по всьому світу. Тут з'являються нові виклики: мережеві затримки, відмови вузлів, розділення мережі.

Підрозділ 13.4 розповідає історію однієї з найвпливовіших ідей в історії обчислень. MapReduce — це не просто алгоритм, це філософія. Google створив її для обробки петабайтів даних, але геніальність полягає в простоті: будь-яку задачу можна розбити на два етапи — map (перетворення) та reduce (агрегація).

13.1. Паралельні алгоритми: паралельні сортування

Паралельні сортування: Bitonic Sort

Теоретичні відомості

Bitonic sort є алгоритмом паралельного сортування, розробленим спеціально для ефективної роботи на паралельних архітектурах. Алгоритм базується на концепції бітонічної послідовності та використовує підхід "розділяй і володарюй" для створення впорядкованої послідовності з довільного набору даних.

Бітонічна послідовність - це послідовність чисел, яка спочатку монотонно зростає, а потім монотонно спадає, або може бути циклічно зсунута, щоб мати таку властивість. Формально, послідовність $a_0, a_1, \ldots, a_{n-1}$ є бітонічною, якщо існує індекс i такий, що:

$$a_0 \leq a_1 \leq \ldots \leq a_i \geq a_{i+1} \geq \ldots \geq a_{n-1}$$

або послідовність може бути циклічно зсунута для отримання такої форми.

Ключовою властивістю бітонічних послідовностей є те, що їх можна ефективно розділити на дві бітонічні підпослідовності, де всі елементи однієї підпослідовності менші або рівні всім елементам іншої. Ця властивість лежить в основі алгоритму Bitonic sort.

Алгоритм працює у два основні етапи.

1. Перший етап полягає у побудові бітонічної послідовності з вхідних даних.

2. Другий етап використовує процедуру бітонічного злиття для перетворення бітонічної послідовності у відсортовану.

Важливою особливістю алгоритму є те, що він працює лише з масивами, розмір яких є степенем двійки.

Основною операцією в алгоритмі є операція порівняння та обміну, яка виконується між парами елементів. Ці операції можуть виконуватися паралельно для всіх пар на кожному кроці, що робить алгоритм надзвичайно ефективним на паралельних архітектурах.

Бітонічне злиття

Процедура бітонічного злиття є центральною частиною алгоритму. Вона приймає бітонічну послідовність і перетворює її на дві бітонічні підпослідовності, де всі елементи першої підпослідовності менші за всі елементи другої.

Для послідовності довжиною n процедура працює наступним чином:

- Порівнюємо елемент з індексом i з елементом з індексом $i + n/2$ для всіх i від 0 до $n/2 - 1$

- Якщо сортуємо за зростанням, то менший елемент розміщується в першій половині, а більший - у другій

- Якщо сортуємо за спаданням, то навпаки.

Після цього кроку отримуємо дві бітонічні підпослідовності довжиною $n/2$ кожна. Процедура рекурсивно застосовується до кожної підпослідовності, поки не досягнемо підпослідовностей довжиною 1.

Побудова бітонічної послідовності

Для побудови бітонічної послідовності з довільного масиву використовується рекурсивний підхід. Масив розділяється навпіл, перша половина сортується за зростанням, а друга - за спаданням. Це створює бітонічну послідовність. Процес рекурсивно застосовується до кожної половини.

Наведемо алгоритм Bitonic sort:

```
function bitonicSort(arr, low, count, direction):
    if count > 1:
        k = count / 2

        // Рекурсивно сортуємо першу половину за зростанням
        bitonicSort(arr, low, k, 1)
```

```
        // Рекурсивно сортуємо другу половину за спаданням
        bitonicSort(arr, low + k, k, 0)

        // Злиття двох половин
        bitonicMerge(arr, low, count, direction)

function bitonicMerge(arr, low, count, direction):
    if count > 1:
        k = count / 2

        for i from low to low + k - 1:
            compareAndSwap(arr, i, i + k, direction)

        // Рекурсивно застосовуємо злиття до обох половин
        bitonicMerge(arr, low, k, direction)
        bitonicMerge(arr, low + k, k, direction)

function compareAndSwap(arr, i, j, direction):
    if direction == 1:  // Зростання
        if arr[i] > arr[j]:
            swap(arr[i], arr[j])
    else:  // Спадання
        if arr[i] < arr[j]:
            swap(arr[i], arr[j])
```

Параметр `direction` визначає напрямок сортування: 1 для зростання, 0 для спадання. Початковий виклик для сортування всього масиву за зростанням буде: `bitonicSort(arr, 0, n, 1)`, де `n` - розмір масиву.

Розглянемо детальний приклад сортування масиву з 8 елементів:

[3, 7, 4, 8, 6, 2, 1, 5].

Крок 1: Побудова бітонічних послідовностей довжиною 2

Розбиваємо масив на пари і сортуємо кожну пару: перші пари за зростанням, другі - за спаданням.

- Пара (3, 7): сортуємо за зростанням → [3, 7]
- Пара (4, 8): сортуємо за спаданням → [8, 4]
- Пара (6, 2): сортуємо за зростанням → [2, 6]
- Пара (1, 5): сортуємо за спаданням → [5, 1]

Результат: [3, 7, 8, 4, 2, 6, 5, 1]

Крок 2: Побудова бітонічних послідовностей довжиною 4

Тепер маємо дві послідовності довжиною 4.

Перша [3, 7, 8, 4] вже є бітонічною (зростає, потім спадає).

Друга [2, 6, 5, 1] також бітонічна.

Застосовуємо бітонічне злиття:

- Для першої четвірки (сортування за зростанням):
    - Порівнюємо 3 і 8: 3 < 8, залишаємо
    - Порівнюємо 7 і 4: 7 > 4, міняємо → [3, 4, 8, 7]
    - Рекурсивно застосовуємо до [3, 4] та [8, 7]
    - Результат: [3, 4, 7, 8]
- Для другої четвірки (сортування за спаданням):
    - Порівнюємо 2 і 5: 2 < 5, міняємо → [5, 6, 2, 1]
    - Порівнюємо 6 і 1: 6 > 1, залишаємо
    - Рекурсивно застосовуємо
    - Результат: [6, 5, 2, 1]

Маємо: [3, 4, 7, 8, 6, 5, 2, 1]

Крок 3: Фінальне злиття

Тепер весь масив є бітонічною послідовністю. Застосовуємо фінальне бітонічне злиття:

- Порівнюємо елементи з відстанню 4:
    - 3 і 6: 3 < 6, залишаємо
    - 4 і 5: 4 < 5, залишаємо
    - 7 і 2: 7 > 2, міняємо
    - 8 і 1: 8 > 1, міняємо

Отримуємо: [3, 4, 2, 1, 6, 5, 7, 8]

Продовжуємо рекурсивно застосовувати злиття до підмасивів, поки не отримаємо повністю відсортований масив: [1, 2, 3, 4, 5, 6, 7, 8].

Аналіз складності

Часова складність алгоритму Bitonic sort становить $O(n\log^2 n)$ для послідовної реалізації. Однак при паралельній реалізації з n процесорами складність зменшується до $O(\log^2 n)$, оскільки всі порівняння на кожному рівні можуть виконуватися одночасно.

Глибина рекурсії алгоритму становить $\log n$, і на кожному рівні рекурсії виконується $O(\log n)$ кроків злиття. Загальна кількість порівнянь дорівнює $\frac{n\log^2 n}{2}$.

Просторова складність алгоритму становить $O(\log n)$ для рекурсивного стеку викликів, що робить його ефективним з точки зору використання пам'яті.

Область застосування

Bitonic sort знаходить широке застосування в різних областях, де важлива паралельна обробка даних:

- GPU-обчислення: Алгоритм ідеально підходить для реалізації на графічних процесорах завдяки своїй регулярній структурі та можливості паралельного виконання операцій порівняння-обміну.

- Мережеві процесори: Використовується в спеціалізованих мережевих пристроях для швидкого сортування пакетів даних.

- FPGA та ASIC: Апаратна реалізація алгоритму дозволяє створювати надшвидкі сортувальні модулі для обробки потоків даних у реальному часі.

- Системи реального часу: Передбачуваність часу виконання робить алгоритм придатним для систем з жорсткими часовими обмеженнями.

- Наукові обчислення: Використовується в задачах молекулярної динаміки, обробки зображень та інших областях, де потрібне швидке сортування великих обсягів даних.

Приклади застосування

Приклад 1: Сортування пікселів у обробці зображень

При реалізації медіанних фільтрів для зображень часто потрібно сортувати значення пікселів у вікні. Bitonic sort дозволяє ефективно виконувати це на GPU, обробляючи тисячі вікон одночасно.

Приклад 2: База даних з паралельною обробкою

У системах баз даних, що використовують паралельну обробку запитів, Bitonic sort може застосовуватися для операцій сортування та злиття при виконанні JOIN-операцій.

Приклад 3: Обробка потокових даних

У системах обробки потокових даних, таких як фінансові торгові системи, Bitonic sort використовується для підтримки відсортованих буферів даних, що постійно оновлюються.

Переваги та недоліки

Переваги:

- Висока ефективність на паралельних архітектурах завдяки регулярній структурі обчислень

- Передбачуваний час виконання - кількість операцій не залежить від вхідних даних

- Простота реалізації на апаратному рівні

- Відсутність необхідності в синхронізації між потоками на одному рівні

- Ефективне використання кешу процесора завдяки локальності даних

Недоліки:

- Обмеження на розмір масиву - працює лише зі степенями двійки

- Гірша асимптотична складність порівняно з оптимальними послідовними алгоритмами

- Не є адаптивним - виконує однакову кількість операцій незалежно від початкового порядку даних

- Потребує додаткових операцій для обробки масивів довільного розміру

- Менш ефективний на послідовних архітектурах порівняно зі швидкими послідовними алгоритмами

Контрольні запитання

1. Що таке бітонічна послідовність і які її основні властивості?

2. Яка основна ідея алгоритму Bitonic sort?

3. Чому алгоритм працює тільки з масивами розміром 2^k?

4. Яка часова складність алгоритму при послідовному та паралельному виконанні?

5. Як працює процедура бітонічного злиття?

6. У чому полягає різниця між побудовою бітонічної послідовності та її сортуванням?

7. Які переваги має Bitonic sort для GPU-обчислень?

8. Чому алгоритм називається “бітонічним”?

9. Як можна адаптувати алгоритм для масивів довільного розміру?

10. У яких випадках Bitonic sort ефективніший за інші алгоритми сортування?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму паралельного сортування Bitonic sort

2. Мета

Вивчити принципи роботи алгоритму Bitonic sort, реалізувати його послідовну та паралельну версії, дослідити ефективність алгоритму на різних розмірах даних та кількості потоків.

3. Завдання

- Реалізувати послідовну версію алгоритму Bitonic sort

- Реалізувати паралельну версію з використанням OpenMP або потоків

- Створити функцію для генерації тестових даних різного розміру

- Провести порівняльний аналіз продуктивності послідовної та паралельної версій

- Візуалізувати процес сортування для невеликих масивів

- Дослідити вплив кількості потоків на швидкість виконання

4. Опис

Лабораторна робота передбачає поетапну реалізацію алгоритму Bitonic sort. Спочатку необхідно реалізувати базові функції порівняння та обміну, потім функції бітонічного злиття та сортування. Паралельна версія повинна розподіляти операції порівняння-обміну між потоками на кожному рівні алгоритму.

5. Порядок виконання

- Створити проект та підключити необхідні бібліотеки

- Реалізувати функцію `compareAndSwap` для порівняння та обміну елементів

- Реалізувати функцію `bitonicMerge` для злиття бітонічної послідовності

- Реалізувати функцію `bitonicSort` для повного сортування

- Створити функцію генерації випадкових масивів розміром 2^k

- Реалізувати паралельну версію з використанням директив OpenMP

- Створити функцію вимірювання часу виконання

- Провести тестування на масивах різного розміру

- Побудувати графіки залежності часу виконання від розміру масиву та кількості потоків

6. Результат

Програма повинна коректно сортувати масиви розміром 2^k, демонструвати прискорення при паралельному виконанні, виводити статистику часу виконання та візуалізувати процес сортування для невеликих масивів.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функція порівняння та обміну елементів

- Функція бітонічного злиття

- Функція бітонічного сортування

- Паралельна реалізація з використанням OpenMP

- Функції тестування та вимірювання продуктивності

2. Продемонструвати роботу додатку:

- Сортування масивів різного розміру

- Порівняння часу виконання послідовної та паралельної версій

- Візуалізацію кроків алгоритму для масиву з 8-16 елементів

- Графіки залежності продуктивності від параметрів

3. Бути готовим відповісти на питання:

- Як працює бітонічне злиття?

- Чому алгоритм ефективний для паралельного виконання?

- Як розподіляється робота між потоками?

- Які фактори впливають на ефективність паралельної версії?

Тести

1. Яка часова складність алгоритму Bitonic sort при послідовному виконанні?

- 1) $O(n\log n)$

- 2) $O(n^2)$

- 3) $O(n\log^2 n)$

- 4) $O(n^2\log n)$

2. Що таке бітонічна послідовність?

- 1) Послідовність, що тільки зростає
- 2) Послідовність, що спочатку зростає, а потім спадає
- 3) Випадкова послідовність чисел
- 4) Послідовність парних чисел

3. З якими розмірами масивів працює стандартна версія Bitonic sort?
- 1) Будь-якими
- 2) Тільки парними
- 3) Тільки степенями двійки
- 4) Тільки простими числами

4. Яка операція є основною в алгоритмі Bitonic sort?
- 1) Пошук мінімуму
- 2) Порівняння та обмін
- 3) Злиття масивів
- 4) Розбиття на підмасиви

5. Яка часова складність паралельної версії Bitonic sort з n процесорами?
- 1) $O(\log n)$
- 2) $O(\log^2 n)$
- 3) $O(n)$
- 4) $O(1)$

Відповіді:

1-3,

2-2,

3-3,

4-2,

5-2

Тести підвищеної складності

1. Скільки порівнянь виконує Bitonic sort для масиву з n елементів?
- 1) $n\log n$
- 2) $\frac{n\log n}{2}$

- 3) $\frac{n\log^2 n}{2}$

- 4) $n\log^2 n$

2. Яка глибина рекурсії алгоритму Bitonic sort?

- 1) $\log n$

- 2) $\log^2 n$

- 3) n

- 4) $\sqrt{n}$

3. Як можна адаптувати Bitonic sort для масиву розміром, що не є степенем двійки?

- 1) Неможливо адаптувати

- 2) Доповнити масив елементами до найближчої степені двійки

- 3) Використати іншу версію алгоритму

- 4) Відкинути зайві елементи

4. Яка властивість робить Bitonic sort придатним для апаратної реалізації?

- 1) Мала кількість операцій

- 2) Регулярна структура обчислень

- 3) Використання додаткової пам’яті

- 4) Адаптивність до вхідних даних

5. У якому випадку Bitonic sort виконує мінімальну кількість обмінів?

- 1) Масив вже відсортований

- 2) Масив відсортований у зворотному порядку

- 3) Всі елементи однакові

- 4) Кількість обмінів завжди однакова

6. Яка просторова складність рекурсивної реалізації Bitonic sort?

- 1) $O(1)$

- 2) $O(\log n)$

- 3) $O(n)$

- 4) $O(n\log n)$

7. Чому Bitonic sort не є оптимальним для послідовного виконання?

- 1) Використовує багато пам’яті

- 2) Має гіршу асимптотику ніж оптимальні алгоритми

- 3) Працює тільки з цілими числами

- 4) Нестабільний алгоритм

8. Яка максимальна відстань між елементами, що порівнюються в Bitonic sort?

- 1) 1

- 2) $\log n$

- 3) $n/2$

- 4) $n - 1$

9. Скільки фаз злиття потрібно для сортування масиву з 16 елементів?

- 1) 4

- 2) 8

- 3) 10

- 4) 16

10. Яка особливість Bitonic sort робить його передбачуваним для систем реального часу?

- 1) Мала кількість операцій

- 2) Час виконання не залежить від вхідних даних

- 3) Використання малої кількості пам’яті

- 4) Простота реалізації

Відповіді:

1. 3) Загальна кількість порівнянь дорівнює $\frac{n\log^2 n}{2}$, оскільки на кожному з $\log n$ рівнів виконується $\frac{n\log n}{2}$ порівнянь

2. 1) Глибина рекурсії дорівнює $\log n$, оскільки на кожному рівні розмір підзадач зменшується вдвічі

3. 2) Найпростіший спосіб - доповнити масив максимальними або мінімальними значеннями до найближчої степені двійки

4. 2) Регулярна структура обчислень дозволяє легко реалізувати алгоритм на апаратному рівні

5. 4) Bitonic sort виконує фіксовану кількість порівнянь незалежно від початкового порядку елементів

6. 2) Просторова складність визначається глибиною рекурсивних викликів, яка дорівнює $O(\log n)$

7. 2) Часова складність $O(n\log^2 n)$ гірша за $O(n\log n)$ оптимальних алгоритмів сортування

8. 3) Максимальна відстань досягається на першому кроці злиття і дорівнює $n/2$

9. 3) Для 16 елементів потрібно $1 + 2 + 3 + 4 = 10$ фаз злиття

10. 2) Кількість операцій фіксована і не залежить від розподілу даних, що гарантує передбачуваний час виконання

Завдання для самостійної роботи

1. Реалізуйте модифікацію Bitonic sort для сортування масивів довільного розміру, використовуючи техніку доповнення спеціальними значеннями.

2. Створіть візуалізацію алгоритму Bitonic sort, що показує всі етапи порівнянь та обмінів для масиву з 16 елементів.

3. Реалізуйте версію Bitonic sort для сортування за кількома ключами (multi-key sorting).

4. Дослідіть та реалізуйте оптимізацію, що зменшує кількість порівнянь при сортуванні частково впорядкованих масивів.

5. Створіть CUDA-реалізацію Bitonic sort для GPU та порівняйте її продуктивність з CPU-версією.

6. Реалізуйте гібридний алгоритм, що використовує Bitonic sort для малих підмасивів у алгоритмі швидкого сортування.

7. Розробіть версію алгоритму для сортування даних, що не поміщаються в оперативну пам’ять (external sorting).

8. Створіть бенчмарк для порівняння Bitonic sort з іншими паралельними алгоритмами сортування на різних архітектурах.

9. Реалізуйте мережеву версію Bitonic sort для розподілених систем з використанням MPI.

10. Дослідіть вплив різних стратегій розподілу роботи між потоками на ефективність паралельної версії алгоритму.

Теми лабораторних робіт

1. Базова реалізація Bitonic sort та візуалізація його роботи для навчальних цілей.

2. Паралельна реалізація Bitonic sort з використанням OpenMP та аналіз масштабованості.

3. CUDA-реалізація Bitonic sort для сортування великих масивів на GPU.

4. Порівняльний аналіз Bitonic sort з іншими паралельними алгоритмами сортування.

5. Реалізація апаратного сортувальника на основі Bitonic sort з використанням FPGA.

6. Застосування Bitonic sort для обробки потокових даних у реальному часі.

7. Гібридні алгоритми сортування з використанням Bitonic sort для спеціальних випадків.

8. Оптимізація Bitonic sort для конкретних архітектур процесорів з урахуванням кеш-пам'яті.

9. Розробка бібліотеки паралельного сортування з підтримкою різних типів даних на основі Bitonic sort.

10. Дослідження енергоефективності різних реалізацій Bitonic sort на сучасних процесорах.

Паралельні сортування з використанням префіксних сум

Теоретичні відомості

Паралельне сортування з використанням префіксних сум є потужним підходом до організації ефективної обробки даних на паралельних обчислювальних системах. Цей метод дозволяє розподілити обчислювальне навантаження між декількома процесорами або потоками, використовуючи префіксні суми для координації роботи та забезпечення правильного порядку елементів.

Префіксна сума - це операція, яка для послідовності елементів $a_0, a_1, \ldots, a_{n-1}$ обчислює нову послідовність $s_0, s_1, \ldots, s_{n-1}$, де кожен елемент s_i дорівнює сумі всіх попередніх елементів:

$$s_i = \sum_{j=0}^{i} a_j$$

У контексті паралельного сортування префіксні суми використовуються для визначення позицій елементів у відсортованому масиві. Основна ідея полягає в тому, що замість послідовного порівняння та переміщення

елементів, ми паралельно обчислюємо, скільки елементів повинно стояти перед кожним елементом у відсортованій послідовності.

Алгоритм паралельного сортування з префіксними сумами базується на кількох ключових етапах:

- Розподіл даних: Вхідний масив розбивається на блоки, кожен з яких обробляється окремим процесором або потоком.

- Локальне сортування: Кожен процесор сортує свій блок даних незалежно від інших, використовуючи будь-який ефективний алгоритм сортування.

- Підрахунок рангів: Для кожного елемента визначається його ранг - кількість елементів у всьому масиві, які менші або рівні даному елементу.

- Обчислення префіксних сум: Використовуючи ранги елементів, обчислюються префіксні суми для визначення фінальних позицій елементів.

- Переміщення елементів: Елементи переміщуються на свої фінальні позиції відповідно до обчислених префіксних сум.

Розглянемо алгоритм паралельного сортування для n елементів з використанням p процесорів:

Крок 1: Розподіл даних

Вхідний масив $A[0..n-1]$ розбивається на p блоків розміром приблизно n/p елементів кожен. Процесор P_i отримує блок $A[i \cdot n/p..(i+1) \cdot n/p - 1]$.

Крок 2: Локальне сортування

Кожен процесор P_i сортує свій локальний блок, використовуючи ефективний послідовний алгоритм сортування, наприклад, quicksort або mergesort. Після цього кроку кожен блок впорядкований, але загальний масив ще не відсортований.

Крок 3: Вибірка зразків

Для забезпечення рівномірного розподілу елементів між процесорами, кожен процесор вибирає $p-1$ зразків зі свого відсортованого блоку. Зразки вибираються рівномірно: елементи на позиціях $k \cdot n/p^2$ для $k = 1,2,\ldots,p-1$.

Крок 4: Глобальне сортування зразків

Всі вибрані зразки збираються на одному процесорі (зазвичай P_0), який сортує їх та вибирає $p-1$ глобальних роздільників. Ці роздільники

визначають межі для розподілу елементів між процесорами на фінальному етапі.

Крок 5: Підрахунок елементів для кожного діапазону

Кожен процесор P_i підраховує, скільки елементів з його локального блоку потрапляє в кожен з p діапазонів, визначених глобальними роздільниками. Результатом є матриця підрахунків $C[i,j]$, де $C[i,j]$ - кількість елементів процесора P_i, які належать діапазону j.

Крок 6: Обчислення префіксних сум

Для кожного стовпця матриці C обчислюються префіксні суми:

$$S[i,j] = \sum_{k=0}^{i} C\,[k,j]$$

Ці суми визначають, скільки елементів з процесорів P_0 до P_i потрапляє в діапазон j.

Крок 7: Перерозподіл елементів

Використовуючи обчислені префіксні суми, кожен процесор визначає, куди відправити свої елементи. Елементи, що належать діапазону j, відправляються процесору P_j на позиції, визначені префіксними сумами.

Крок 8: Фінальне локальне сортування

Після перерозподілу кожен процесор P_j містить всі елементи, які належать його діапазону. Процесор виконує фінальне сортування своїх елементів.

Наведемо псевдокод алгоритму

```
function ParallelSortWithPrefixSum(A, n, p)
    // A - вхідний масив
    // n - кількість елементів
    // p - кількість процесорів

    // Крок 1: Розподіл даних
    blockSize = n / p
    localBlock = A[myRank * blockSize .. (myRank + 1) * blockSize - 1
]

    // Крок 2: Локальне сортування
    localSort(localBlock)

    // Крок 3: Вибірка зразків
    samples = []
```

```
    for k = 1 to p-1 do
        samples.append(localBlock[k * blockSize / p])

    // Крок 4: Збір та сортування зразків
    allSamples = gather(samples, root=0)
    if myRank == 0 then
        sort(allSamples)
        splitters = []
        for k = 1 to p-1 do
            splitters.append(allSamples[k * p])
    broadcast(splitters, root=0)

    // Крок 5: Підрахунок елементів для кожного діапазону
    counts = array[p]
    for j = 0 to p-1 do
        counts[j] = countElementsInRange(localBlock, j, splitters)

    // Крок 6: Обчислення префіксних сум
    prefixSums = computePrefixSums(counts)

    // Крок 7: Перерозподіл елементів
    for j = 0 to p-1 do
        elements = getElementsInRange(localBlock, j, splitters)
        send(elements, processor=j, position=prefixSums[myRank][j])

    // Отримання елементів від інших процесорів
    receivedElements = receiveElements()

    // Крок 8: Фінальне сортування
    finalSort(receivedElements)

    return receivedElements
```

Ефективне паралельне обчислення префіксних сум є критичним компонентом алгоритму. Розглянемо алгоритм Blelloch для паралельного обчислення префіксних сум:

```
function ParallelPrefixSum(A, n)
    // Фаза підйому (up-sweep)
    for d = 0 to log2(n) - 1 do
        parallel for k = 0 to n-1 by 2^(d+1) do
            A[k + 2^(d+1) - 1] = A[k + 2^d - 1] + A[k + 2^(d+1) - 1]

    // Встановлення кореня в нуль
    A[n-1] = 0
```

```
    // Фаза спуску (down-sweep)
    for d = log2(n) - 1 downto 0 do
        parallel for k = 0 to n-1 by 2^(d+1) do
            temp = A[k + 2^d - 1]
            A[k + 2^d - 1] = A[k + 2^(d+1) - 1]
            A[k + 2^(d+1) - 1] = temp + A[k + 2^(d+1) - 1]

    return A
```

Область застосування

Паралельне сортування з використанням префіксних сум знаходить широке застосування в різних областях:

- Обробка великих даних: Сортування масивних наборів даних у розподілених системах та кластерах.

- Бази даних: Паралельне виконання операцій сортування та індексування в системах управління базами даних.

- Наукові обчислення: Обробка результатів моделювання та експериментальних даних у фізиці, біології, метеорології.

- Машинне навчання: Підготовка даних для алгоритмів навчання, сортування за ознаками або важливістю.

- Комп'ютерна графіка: Сортування примітивів для алгоритмів рендерингу, обробка точкових хмар.

- Фінансова аналітика: Обробка транзакцій та ринкових даних у реальному часі.

Приклади застосування

Приклад 1: Сортування невеликого масиву

Розглянемо роботу алгоритму на прикладі сортування масиву з 8 елементів використовуючи 2 процесори:

Вхідний масив: `[7, 3, 9, 1, 5, 8, 2, 6]`

Крок 1: Розподіл між процесорами

- Процесор P0: `[7, 3, 9, 1]`

- Процесор P1: `[5, 8, 2, 6]`

Крок 2: Локальне сортування

- Процесор P0: `[1, 3, 7, 9]`

- Процесор P1: `[2, 5, 6, 8]`

Крок 3: Вибірка зразків (по одному з кожного процесора)

- Процесор P0 вибирає: 3

- Процесор P1 вибирає: 5

Крок 4: Визначення глобального роздільника Роздільник

= 4 (середнє між 3 та 5)

Крок 5: Підрахунок елементів

- P0: 2 елементи ⩽ 4 (це 1 та 3), 2 елементи > 4 (це 7 та 9)

- P1: 1 елемент ⩽ 4 (це 2), 3 елементи > 4 (це 5, 6, 8)

Крок 6: Обчислення префіксних сум Для елементів ⩽ 4:

- P0: позиції 0-1

- P1: позиція 2

Для елементів > 4:

- P0: позиції 0-1

- P1: позиції 2-4

Крок 7: Перерозподіл

- P0 отримує: `[1, 3, 2]`

- P1 отримує: `[7, 9, 5, 6, 8]`

Крок 8: Фінальне сортування

- P0: `[1, 2, 3]`

- P1: `[5, 6, 7, 8, 9]`

Результат: `[1, 2, 3, 5, 6, 7, 8, 9]`

Приклад 2: Обчислення префіксних сум

Демонстрація паралельного обчислення префіксних сум для масиву `[3, 1, 7, 0, 4, 1, 6, 3]`:

Фаза підйому:

```
Початковий стан: [3, 1, 7, 0, 4, 1, 6, 3]
d=0: [3, 4, 7, 7, 4, 5, 6, 9]
d=1: [3, 4, 7, 11, 4, 5, 6, 14]
d=2: [3, 4, 7, 11, 4, 5, 6, 25]
```

Встановлення кореня: `[3, 4, 7, 11, 4, 5, 6, 0]`

Фаза спуску:

```
d=2: [3, 4, 7, 0, 4, 5, 6, 11]
d=1: [3, 0, 7, 4, 4, 11, 6, 16]
d=0: [0, 3, 4, 11, 11, 15, 16, 22]
```
Результат: **`[0, 3, 4, 11, 11, 15, 16, 22]`** - префіксні суми вхідного масиву.

Переваги та недоліки

Переваги

- Масштабованість: Алгоритм ефективно масштабується зі збільшенням кількості процесорів, забезпечуючи близьке до лінійного прискорення для великих наборів даних.

- Збалансоване навантаження: Використання зразків та роздільників забезпечує рівномірний розподіл роботи між процесорами.

- Теоретична ефективність: Часова складність $O(n\log n/p + p\log p)$ для n елементів та p процесорів є оптимальною для паралельних алгоритмів сортування.

- Стабільність: Алгоритм можна модифікувати для забезпечення стабільного сортування, зберігаючи відносний порядок рівних елементів.

- Низька комунікаційна складність: Основні обміни даними відбуваються лише під час збору зразків та фінального перерозподілу.

Недоліки

- Складність реалізації: Алгоритм вимагає ретельної координації між процесорами та складної логіки обчислення префіксних сум.

- Накладні витрати для малих даних: Для невеликих масивів накладні витрати на комунікацію можуть перевищувати виграш від паралелізації.

- Вимоги до пам’яті: Необхідність зберігання проміжних результатів та буферів для обміну даними збільшує використання пам’яті.

- Чутливість до розподілу даних: Нерівномірний розподіл значень у вхідних даних може призвести до дисбалансу навантаження.

- Залежність від архітектури: Ефективність алгоритму залежить від характеристик комунікаційної мережі та топології системи.

Контрольні запитання

1. Що таке префіксна сума та яка її роль у паралельному сортуванні?

2. Поясніть основні етапи алгоритму паралельного сортування з використанням префіксних сум.

3. Як вибираються глобальні роздільники та чому це важливо для збалансованого розподілу роботи?

4. Опишіть алгоритм Blelloch для паралельного обчислення префіксних сум.

5. Яка часова складність алгоритму паралельного сортування з префіксними сумами?

6. Які основні переваги паралельного сортування порівняно з послідовними алгоритмами?

7. У яких випадках паралельне сортування може бути неефективним?

8. Як забезпечити стабільність паралельного алгоритму сортування?

9. Поясніть роль фази вибірки зразків у алгоритмі.

10. Які комунікаційні операції є найбільш критичними для продуктивності алгоритму?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація паралельного сортування з використанням префіксних сум

2. Мета

Освоїти принципи паралельного сортування, навчитися реалізовувати алгоритми обчислення префіксних сум та застосовувати їх для ефективного розподілу даних між процесорами в задачах сортування.

3. Завдання

Реалізувати паралельний алгоритм сортування з використанням префіксних сум, який включає: - Розподіл даних між процесорами - Локальне сортування блоків - Вибірку зразків та визначення роздільників - Паралельне обчислення префіксних сум - Перерозподіл елементів між процесорами - Фінальне сортування

4. Опис

Лабораторна робота передбачає реалізацію повного циклу паралельного сортування. Студенти повинні реалізувати як основний алгоритм сортування, так і допоміжні функції для обчислення префіксних сум. Реалізація може бути виконана з використанням MPI, OpenMP або іншої технології паралельного програмування.

Програма повинна включати: - Генерацію або зчитування вхідних даних - Вимірювання часу виконання кожного етапу алгоритму - Перевірку

коректності результату - Порівняння з послідовним алгоритмом сортування

5. Порядок виконання

- Створити функцію генерації тестових даних різного розміру та розподілу

- Реалізувати функцію паралельного обчислення префіксних сум

- Реалізувати основний алгоритм паралельного сортування

- Додати функції для збору статистики та вимірювання продуктивності

- Провести тестування на різних наборах даних та кількості процесорів

- Побудувати графіки залежності часу виконання від розміру даних та кількості процесорів

6. Результат

Результатом виконання лабораторної роботи є: - Працююча програма паралельного сортування - Звіт з результатами тестування та аналізом продуктивності - Графіки прискорення та ефективності паралелізації - Висновки щодо оптимальної кількості процесорів для різних розмірів даних

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функція паралельного обчислення префіксних сум

- Функція вибірки зразків та визначення роздільників

- Функція перерозподілу елементів між процесорами

- Основна функція паралельного сортування

2. Продемонструвати роботу додатку:

- Сортування масивів різного розміру (від 1000 до 1000000 елементів)

- Робота з різною кількістю процесорів (2, 4, 8, 16)

- Вимірювання часу виконання та обчислення прискорення

- Перевірка коректності результату

3. Бути готовим відповісти на питання:

- Як обчислюються префіксні суми в паралельному режимі?

- Як вибір роздільників впливає на збалансованість навантаження?

- Які фактори обмежують масштабованість алгоритму?

- Як можна оптимізувати комунікації між процесорами?

Тести

1. Основна мета використання префіксних сум у паралельному сортуванні:

- 1) Прискорення локального сортування

- 2) Визначення позицій елементів у результуючому масиві

- 3) Зменшення обсягу пам’яті

- 4) Спрощення алгоритму

2. Часова складність паралельного сортування з префіксними сумами:

- 1) O(n)

- 2) O(n log n)

- 3) O(n log n / p + p log p)

- 4) $O(n^2/p)$

3. На якому етапі алгоритму відбувається основний обмін даними між процесорами?

- 1) Локальне сортування

- 2) Обчислення префіксних сум

- 3) Перерозподіл елементів

- 4) Вибірка зразків

4. Скільки зразків вибирає кожен процесор для p процесорів?

- 1) 1

- 2) p

- 3) p-1

- 4) 2p

5. Що визначають глобальні роздільники в алгоритмі?

- 1) Порядок сортування

- 2) Межі діапазонів для розподілу елементів

- 3) Кількість ітерацій

- 4) Розмір блоків

Відповіді:

1-2,

2-3,

3-3,

4-3,

5-2

Тести підвищеної складності

1. У алгоритмі Blelloch для паралельного обчислення префіксних сум, яка операція виконується на фазі підйому?

- 1) Обчислення часткових сум сусідніх елементів

- 2) Обчислення різниць між елементами

- 3) Обчислення сум елементів на відстані степеня двійки

- 4) Копіювання елементів

2. Яка основна проблема може виникнути при поганому виборі роздільників?

- 1) Алгоритм не завершиться

- 2) Результат буде неправильним

- 3) Дисбаланс навантаження між процесорами

- 4) Збільшення використання пам’яті

3. Для забезпечення стабільності паралельного сортування необхідно:

- 1) Використовувати стабільний локальний алгоритм сортування

- 2) Зберігати початкові індекси елементів

- 3) Обидва варіанти a та b

- 4) Використовувати більше процесорів

4. Яка комунікаційна складність алгоритму паралельного сортування?

- 1) O(n)

- 2) O(n/p)

- 3) $O(p^2)$

- 4) O(n/p + p log p)

5. При якому розподілі вхідних даних алгоритм працює найгірше?

- 1) Рівномірний розподіл

- 2) Нормальний розподіл

- 3) Всі елементи однакові

- 4) Дуже нерівномірний розподіл з кластерами значень

6. Скільки фаз має алгоритм Blelloch для обчислення префіксних сум?

- 1) 1

- 2) 2

- 3) 3

- 4) log n

7. Яка просторова складність паралельного алгоритму сортування?

- 1) O(n)

- 2) O(n/p)

- 3) $O(n + p^2)$

- 4) O(n log n)

8. Для якого типу паралельних систем алгоритм найбільш ефективний?

- 1) Системи зі спільною пам’яттю

- 2) Системи з розподіленою пам’яттю

- 3) Гібридні системи

- 4) Однаково ефективний для всіх

9. Який фактор найбільше впливає на вибір оптимальної кількості процесорів?

- 1) Розмір вхідних даних

- 2) Тип процесорів

- 3) Мова програмування

- 4) Операційна система

10. Як можна мінімізувати комунікаційні витрати в алгоритмі?

- 1) Використати більше процесорів

- 2) Збільшити розмір локальних блоків

- 3) Використати асинхронні комунікації

- 4) Зменшити кількість зразків

Відповіді:

1. 3) Обчислюються суми елементів, що знаходяться на відстані степеня двійки, створюючи дерево часткових сум

2. 3) Погані роздільники призводять до того, що деякі процесори отримують значно більше елементів, ніж інші

3. 3) Необхідно як використовувати стабільне локальне сортування, так і зберігати початкові індекси для правильного впорядкування рівних елементів

4. 4) Комунікації відбуваються при зборі зразків $O(p \log p)$ та перерозподілі елементів $O(n/p)$

5. 4) Кластери значень призводять до того, що багато елементів потрапляють в один діапазон

6. 2) Алгоритм має дві фази - підйом (up-sweep) та спуск (down-sweep)

7. 3) Потрібна пам’ять для зберігання даних $O(n)$ та матриці підрахунків $O(p^2)$

8. 2) Алгоритм оптимізований для систем з розподіленою пам’яттю, де кожен процесор має локальну пам’ять

9. 1) Розмір даних визначає, чи варто використовувати паралелізацію та скільки процесорів буде оптимально

10. 3) Асинхронні комунікації дозволяють перекривати обчислення з передачею даних

Завдання для самостійної роботи

1. Реалізуйте модифікацію алгоритму для сортування даних, що не поміщаються в оперативну пам’ять, використовуючи зовнішню пам’ять.

2. Розробіть адаптивну версію алгоритму, яка автоматично вибирає оптимальну кількість процесорів залежно від розміру даних.

3. Реалізуйте гібридну версію, що поєднує MPI для міжвузлових комунікацій та OpenMP для внутрішньовузлової паралелізації.

4. Створіть візуалізацію роботи алгоритму, що показує розподіл елементів між процесорами на кожному етапі.

5. Дослідіть вплив різних стратегій вибору зразків на збалансованість навантаження та загальну продуктивність.

6. Реалізуйте версію алгоритму для сортування структур даних за кількома ключами з підтримкою стабільності.

7. Розробіть механізм динамічного балансування навантаження для випадків сильно нерівномірного розподілу даних.

8. Створіть бенчмарк для порівняння різних паралельних алгоритмів сортування на різних архітектурах.

9. Реалізуйте відмовостійку версію алгоритму, що може продовжити роботу при відмові одного з процесорів.

10. Дослідіть можливість використання GPU для прискорення обчислення префіксних сум та локального сортування.

Теми лабораторних робіт

1. Базова реалізація паралельного сортування з префіксними сумами для систем зі спільною пам’яттю.

2. Реалізація розподіленого паралельного сортування з використанням MPI.

3. Оптимізація алгоритму для неоднорідних обчислювальних кластерів.

4. Паралельне сортування рядків змінної довжини з використанням префіксних сум.

5. Застосування паралельного сортування для обробки потокових даних у реальному часі.

6. Реалізація паралельного radix sort з використанням префіксних сум.

7. Паралельне сортування з компресією даних для зменшення комунікаційних витрат.

8. Інтеграція паралельного сортування в систему управління базами даних.

9. Реалізація паралельного сортування для обробки геопросторових даних.

10. Розробка фреймворку для автоматичної паралелізації алгоритмів сортування.

13.2. Lock-free/Wait-free структури

Черга Майкла-Скотта

Теоретичні відомості

Черга Майкла-Скотта є одним з найефективніших алгоритмів реалізації неблокуючої черги для багатопотокових систем. Цей алгоритм був розроблений Мередітом Майклом та Майклом Скоттом у 1996 році і став

стандартом де-факто для реалізації lock-free черг у сучасних системах паралельного програмування.

Основна ідея алгоритму полягає у використанні атомарної операції compare-and-swap (CAS) для забезпечення безпечного доступу до черги з боку кількох потоків одночасно без використання традиційних механізмів блокування. Черга реалізується як однозв'язний список з двома вказівниками: Head (голова) вказує на перший елемент черги, а Tail (хвіст) вказує на останній елемент.

Структура даних

Черга Майкла-Скотта складається з вузлів, кожен з яких містить значення та вказівник на наступний вузол:

```
struct Node {
    value: T
    next: pointer to Node
}

struct Queue {
    Head: pointer to Node
    Tail: pointer to Node
}
```

Важливою особливістю є те, що черга завжди містить принаймні один вузол - фіктивний вузол (dummy node), який спрощує реалізацію операцій. При ініціалізації черги створюється цей фіктивний вузол, на який вказують обидва вказівники Head і Tail.

Атомарна операція Compare-And-Swap

Операція CAS є фундаментальною для роботи алгоритму. Вона атомарно порівнює значення в пам'яті з очікуваним значенням і, якщо вони співпадають, замінює його на нове значення. Псевдокод операції CAS виглядає таким чином:

```
function CAS(location, expected, new_value)
    atomically:
        if *location == expected then
            *location = new_value
            return true
        else
            return false
```

Ця операція гарантує, що зміна значення відбудеться тільки якщо жоден інший потік не змінив його з моменту останнього читання.

Операція Enqueue

Операція додавання елемента до черги працює наступним чином: новий вузол додається після поточного хвоста черги, після чого вказівник Tail переміщується на новий вузол. Алгоритм враховує можливість того, що інші потоки можуть одночасно модифікувати чергу.

```
function Enqueue(Q: pointer to Queue, value: T)
    node = new Node()
    node.value = value
    node.next = null

    while true do
        tail = Q.Tail
        next = tail.next

        // Перевіряємо, чи не змінився хвіст поки ми читали
        if tail == Q.Tail then
            if next == null then
                // Намагаємося додати новий вузол
                if CAS(&tail.next, null, node) then
                    // Успішно додали, намагаємося оновити Tail
                    CAS(&Q.Tail, tail, node)
                    break
            else
                // Хвіст не вказує на останній вузол,
                // намагаємося його просунути
                CAS(&Q.Tail, tail, next)
```

Алгоритм використовує цикл, який продовжується доки операція не завершиться успішно. На кожній ітерації перевіряється стан черги та робиться спроба виконати необхідну операцію. Якщо інший потік змінив чергу, операція CAS поверне false, і цикл повториться.

Операція Dequeue

Операція видалення елемента з черги видаляє вузол, на який вказує Head, і повертає його значення. Фіктивний вузол ніколи не видаляється - замість цього Head переміщується на наступний вузол.

```
function Dequeue(Q: pointer to Queue): T or empty
    while true do
        head = Q.Head
        tail = Q.Tail
        next = head.next

        // Перевіряємо консистентність
```

```
        if head == Q.Head then
            if head == tail then
                if next == null then
                    // Черга порожня
                    return empty
                else
                    // Tail відстає, намагаємося його просунути
                    CAS(&Q.Tail, tail, next)
            else
                // Читаємо значення перед CAS
                value = next.value
                // Намагаємося просунути Head
                if CAS(&Q.Head, head, next) then
                    // Видаляємо старий фіктивний вузол
                    delete head
                    return value
```

Важливо відзначити, що алгоритм перевіряє особливий випадок, коли Head і Tail вказують на один вузол. Це може означати або що черга порожня (якщо next == null), або що операція Enqueue була перервана і Tail потребує оновлення.

Механізм допомоги

Унікальною особливістю алгоритму Майкла-Скотта є механізм допомоги, коли один потік може завершити операцію, розпочату іншим потоком. Це видно в обох операціях, де потоки намагаються просунути вказівник Tail, якщо виявляють, що він відстає. Цей механізм забезпечує прогрес алгоритму навіть якщо деякі потоки зупиняються або працюють повільно.

Ініціалізація черги

При створенні черги необхідно створити початковий фіктивний вузол:

```
function InitQueue(): Queue
    node = new Node()
    node.next = null
    Q = new Queue()
    Q.Head = node
    Q.Tail = node
    return Q
```

Властивості алгоритму

Алгоритм Майкла-Скотта має кілька важливих властивостей:

- Lock-free: Алгоритм гарантує, що принаймні один потік завжди зробить прогрес за скінченну кількість кроків, навіть якщо інші потоки зупиняться.

- Лінеаризованість: Кожна операція має точку лінеаризації - момент, коли операція логічно відбувається. Для Enqueue це успішний CAS, що додає новий вузол. Для Dequeue це успішний CAS, що переміщує Head.

- Коректність: Алгоритм гарантує, що черга завжди залишається в консистентному стані, навіть при одночасному доступі багатьох потоків.

Область застосування

Черга Майкла-Скотта широко використовується в сучасних системах паралельного програмування:

- Операційні системи: Для реалізації черг планувальника, обробки переривань та міжпроцесної комунікації.

- Системи баз даних: У механізмах управління транзакціями та буферизації запитів.

- Мережеві сервери: Для обробки вхідних з'єднань та запитів у високонавантажених системах.

- Ігрові движки: Для передачі повідомлень між різними підсистемами без блокувань.

- Фреймворки паралельного програмування: Як базовий примітив для побудови складніших структур даних.

- Системи реального часу: Де важлива передбачуваність часу виконання операцій.

Приклади застосування

Приклад 1: Проста демонстрація роботи

Розглянемо послідовність операцій з чергою, яку виконують три потоки одночасно:

Початковий стан: черга містить фіктивний вузол

```
Head -> [dummy] <- Tail
```

Потік 1 виконує Enqueue(A):

1. Створює новий вузол з значенням A
2. Намагається приєднати його до dummy.next
3. Успішно виконує CAS(&dummy.next, null, nodeA)
4. Оновлює Tail

Стан після операції:

```
Head -> [dummy] -> [A] <- Tail
```

Потік 2 виконує Enqueue(B) одночасно з Потоком 3, який виконує Dequeue():

Потік 2:

1. Читає tail = [A]
2. Перевіряє tail.next == null
3. Намагається CAS(&A.next, null, nodeB)

Потік 3:

1. Читає head = [dummy]
2. Читає next = [A]
3. Намагається CAS(&Head, dummy, A)

Якщо Потік 3 завершиться першим:

```
Head -> [A] -> [B] <- Tail
```

Старий dummy вузол видаляється, функція повертає значення A.

Приклад 2: Механізм допомоги

Розглянемо ситуацію, коли операція Enqueue була перервана:

Початковий стан після незавершеної операції Enqueue(C):

```
Head -> [A] -> [B] -> [C]
                 ^
                Tail
```

Tail вказує на [B], хоча [C] вже приєднаний. Тепер Потік 1 намагається виконати Enqueue(D):

1. Читає tail = [B]
2. Читає tail.next = [C] (не null!)
3. Розуміє, що Tail відстає
4. Виконує CAS(&Q.Tail, [B], [C]) - допомагає завершити попередню операцію
5. На наступній ітерації успішно додає [D]

Фінальний стан:

```
Head -> [A] -> [B] -> [C] -> [D] <- Tail
```

Приклад 3: Обробка порожньої черги

Коли черга порожня і кілька потоків намагаються виконати Dequeue:

Початковий стан:

```
Head -> [dummy] <- Tail
```

Потік 1 і Потік 2 одночасно виконують Dequeue():

- Обидва читають head == tail
- Обидва читають next == null
- Обидва повертають “empty”

Черга залишається в консистентному стані, обидва потоки коректно визначають, що черга порожня.

Переваги та недоліки

Переваги

- Висока продуктивність: Відсутність блокувань дозволяє досягти високої пропускної здатності в багатопотокових системах.

- Масштабованість: Продуктивність добре масштабується зі збільшенням кількості потоків.

- Відсутність deadlock: Оскільки немає блокувань, неможливі взаємні блокування.

- Прогрес системи: Lock-free гарантія забезпечує, що система завжди робить прогрес.

- Низька латентність: Операції мають передбачуваний час виконання без непередбачуваних затримок на блокуваннях.

- Стійкість до збоїв: Якщо потік зупиняється, інші потоки продовжують працювати.

Недоліки

- Складність реалізації: Алгоритм складніший за версії з блокуваннями і вимагає глибокого розуміння.

- Витрати пам’яті: Необхідність у фіктивному вузлі та можливість ABA-проблеми можуть збільшити витрати пам’яті.

- Залежність від апаратури: Вимагає апаратної підтримки атомарних операцій CAS.

- Можливість livelock: У випадку високої конкуренції потоки можуть довго повторювати спроби.

- Складність налагодження: Помилки в lock-free алгоритмах важко відтворити та виправити.

- Обмежена переносимість: Різні архітектури можуть мати різну семантику атомарних операцій.

Контрольні запитання

1. Яка роль фіктивного вузла в черзі Майкла-Скотта?

2. Поясніть механізм допомоги в алгоритмі та навіщо він потрібен.

3. Що таке точка лінеаризації для операцій Enqueue та Dequeue?

4. Чому алгоритм використовує цикли в обох операціях?

5. Як алгоритм обробляє ситуацію, коли Tail відстає від реального кінця черги?

6. Які гарантії прогресу надає алгоритм Майкла-Скотта?

7. Чому операція Dequeue перевіряє рівність Head і Tail?

8. Як забезпечується атомарність операцій в алгоритмі?

9. Які переваги має lock-free підхід порівняно з використанням мьютексів?

10. Що станеться, якщо кілька потоків одночасно намагаються виконати Enqueue?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та аналіз черги Майкла-Скотта

2. Мета

Вивчити принципи роботи lock-free структур даних, освоїти використання атомарних операцій, реалізувати неблокуючу чергу Майкла-Скотта та проаналізувати її продуктивність у багатопотоковому середовищі.

3. Завдання

Реалізувати чергу Майкла-Скотта з підтримкою операцій Enqueue та Dequeue, провести тестування коректності роботи в багатопотоковому середовищі, виміряти продуктивність та порівняти з реалізацією на основі мьютексів.

4. Опис

Лабораторна робота передбачає поетапну реалізацію всіх компонентів черги Майкла-Скотта. Студенти повинні реалізувати структуру вузла, операції CAS, алгоритми Enqueue та Dequeue. Особливу увагу слід приділити обробці граничних випадків та забезпеченню коректності в умовах високої конкуренції між потоками.

5. Порядок виконання

- Реалізувати структури Node та Queue
- Створити функцію ініціалізації черги з фіктивним вузлом
- Реалізувати операцію Enqueue з усіма перевірками
- Реалізувати операцію Dequeue з обробкою порожньої черги
- Створити тестову програму з кількома потоками
- Реалізувати альтернативну версію черги з мьютексами
- Провести вимірювання продуктивності для різної кількості потоків
- Створити стрес-тести для перевірки коректності

6. Результат

Працююча реалізація lock-free черги, набір тестів для перевірки коректності, графіки порівняння продуктивності з версією на мьютексах, аналіз результатів з висновками про ефективність алгоритму в різних сценаріях використання.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Реалізація атомарної операції CAS
- Функція Enqueue з обробкою всіх випадків
- Функція Dequeue з механізмом допомоги
- Тестова програма для багатопотокового середовища

2. Продемонструвати роботу додатку:

- Коректна робота з одним потоком
- Робота з кількома потоками-виробниками
- Робота з кількома потоками-споживачами
- Змішаний сценарій з виробниками та споживачами
- Графіки продуктивності

3. Бути готовим відповісти на питання:

- Як працює механізм допомоги?
- Де знаходяться точки лінеаризації операцій?
- Як забезпечується lock-free гарантія?

- Які переваги дає використання фіктивного вузла?

Тести

1. Основна мета використання CAS в алгоритмі Майкла-Скотта:
- 1) Прискорення читання даних
- 2) Атомарна модифікація вказівників без блокувань
- 3) Зменшення використання пам’яті
- 4) Спрощення алгоритму
2. Фіктивний вузол в черзі Майкла-Скотта:
- 1) Зберігає кількість елементів
- 2) Використовується для синхронізації
- 3) Спрощує реалізацію та завжди присутній
- 4) Видаляється при першому Dequeue
3. Механізм допомоги спрацьовує коли:
- 1) Черга порожня
- 2) Tail відстає від реального кінця черги
- 3) Head дорівнює Tail
- 4) Виникає помилка
4. Lock-free гарантія означає:
- 1) Відсутність будь-яких блокувань
- 2) Всі потоки завершаться одночасно
- 3) Принаймні один потік зробить прогрес
- 4) Операції виконуються миттєво
5. Точка лінеаризації для Enqueue:
- 1) Створення нового вузла
- 2) Успішний CAS для додавання вузла
- 3) Оновлення Tail
- 4) Початок функції

Відповіді:

1-2,

2-3,

3-2,

4-3,

5-2

Тести підвищеної складності

1. Чому в Dequeue перевіряється рівність Head і Tail?

- 1) Для оптимізації швидкості

- 2) Щоб уникнути видалення фіктивного вузла

- 3) Для визначення порожньої черги або необхідності допомоги

- 4) Це необов’язкова перевірка

2. Яка операція може не завершитись після успішного CAS в Enqueue?

- 1) Створення вузла

- 2) Приєднання вузла до черги

- 3) Оновлення значення

- 4) Оновлення Tail

3. Скільки разів може виконатись цикл в Enqueue в найгіршому випадку?

- 1) 1 раз

- 2) 2 рази

- 3) O(n) разів

- 4) Необмежену кількість разів

4. Що станеться якщо потік зупиниться після успішного CAS в tail.next але до оновлення Tail?

- 1) Черга стане некоректною

- 2) Інші потоки допоможуть завершити операцію

- 3) Операція буде скасована

- 4) Виникне deadlock

5. Чому алгоритм читає значення до виконання CAS в Dequeue?

- 1) Для оптимізації

- 2) Щоб уникнути читання видаленого вузла

- 3) Це помилка в алгоритмі

- 4) Для збереження порядку

6. Яка складність операцій Enqueue та Dequeue в середньому випадку?

- 1) O(1)

- 2) O(log n)

- 3) O(n)

- 4) $O(n^2)$

7. Що таке ABA-проблема в контексті lock-free алгоритмів?

- 1) Помилка компіляції

- 2) Ситуація коли вказівник змінився і повернувся до початкового значення

- 3) Проблема з пам’яттю

- 4) Тип винятку

8. Скільки CAS операцій виконується в успішному Enqueue в кращому випадку?

- 1) 1

- 2) 2

- 3) 3

- 4) 4

9. Чому черга Майкла-Скотта краще масштабується ніж черга з глобальним блокуванням?

- 1) Використовує менше пам’яті

- 2) Потоки не чекають на блокуваннях

- 3) Простіша реалізація

- 4) Швидші операції

10. Що гарантує лінеаризованість алгоритму?

- 1) Використання CAS

- 2) Наявність фіктивного вузла

- 3) Кожна операція має точку коли вона логічно відбувається

- 4) Відсутність блокувань

Відповіді:

1. 3) Ця перевірка дозволяє визначити чи черга порожня (next==null) або Tail потребує оновлення

2. 4) Оновлення Tail може не відбутись якщо інший потік вже його оновив
3. 4) Теоретично цикл може виконуватись необмежено при постійній конкуренції

4. 2) Механізм допомоги дозволяє іншим потокам завершити операцію оновлення Tail

5. 2) Після CAS вузол може бути видалений іншим потоком, тому значення читається заздалегідь

6. 1) В середньому операції виконуються за константний час

7. 2) Коли вказівник змінюється з A на B, потім знову на A, CAS не помітить зміни

8. 2) Один CAS для приєднання вузла, другий для оновлення Tail (може не виконатись)

9. 2) Відсутність блокувань дозволяє потокам працювати паралельно без очікування

10. 3) Точка лінеаризації визначає момент коли операція логічно завершується в глобальному порядку

Завдання для самостійної роботи

1. Реалізуйте версію черги Майкла-Скотта з підрахунком кількості елементів без використання додаткових блокувань.

2. Модифікуйте алгоритм для підтримки пріоритетів елементів, зберігаючи lock-free властивості.

3. Створіть інструмент візуалізації роботи черги в реальному часі для навчальних цілей.

4. Реалізуйте механізм обмеження розміру черги з блокуванням виробників при переповненні.

5. Дослідіть вплив false sharing на продуктивність та реалізуйте оптимізації для його уникнення.

6. Розробіть версію алгоритму з підтримкою batch операцій для додавання/видалення кількох елементів.

7. Реалізуйте wait-free версію черги та порівняйте її характеристики з lock-free версією.

8. Створіть адаптивну версію, яка перемикається між lock-free та blocking режимами залежно від навантаження.

9. Реалізуйте розподілену версію черги Майкла-Скотта для використання в кластерних системах.

10. Дослідіть можливість використання hardware transactional memory для спрощення алгоритму.

Теми лабораторних робіт

1. Базова реалізація черги Майкла-Скотта та аналіз її коректності.

2. Оптимізація черги для NUMA архітектур з урахуванням локальності даних.

3. Реалізація lock-free стеку на основі принципів алгоритму Майкла-Скотта.

4. Розробка універсального фреймворку для тестування lock-free структур даних.

5. Порівняльний аналіз різних lock-free алгоритмів черг.

6. Реалізація черги Майкла-Скотта з підтримкою персистентності для збереження стану.

7. Інтеграція черги в систему акторів для передачі повідомлень.

8. Розробка профайлера для аналізу конкуренції в lock-free структурах.

9. Реалізація гібридної черги з адаптивним вибором між різними алгоритмами.

10. Створення бібліотеки lock-free структур даних з чергою Майкла-Скотта.

Стек Трейбера

Теоретичні відомості

Стек Трейбера є класичним прикладом lock-free структури даних, розробленої Р. Кентом Трейбером у 1986 році. Це один з найпростіших і водночас найефективніших алгоритмів реалізації неблокуючого стеку для багатопотокових систем. Алгоритм використовує атомарну операцію compare-and-swap (CAS) для забезпечення безпечного доступу до стеку з боку кількох потоків без використання традиційних механізмів блокування.

Основні принципи роботи

Стек Трейбера реалізується як однозв’язний список, де вершина стеку представлена одним вказівником Top, який вказує на верхній елемент. Кожен вузол стеку містить дані та вказівник на наступний елемент.

Ключова ідея полягає в тому, що всі операції зі стеком виконуються через атомарну модифікацію єдиного вказівника Top.

Структура даних стеку виглядає наступним чином:

```
struct Node {
    value: T        // значення, що зберігається в вузлі
    next: pointer to Node  // вказівник на наступний вузол
}

struct Stack {
    top: pointer to Node   // вказівник на вершину стеку
}
```

Атомарна операція Compare-And-Swap

Операція CAS є фундаментальною для роботи стеку Трейбера. Вона атомарно порівнює поточне значення в пам'яті з очікуваним значенням і, якщо вони рівні, замінює його на нове значення. Ця операція гарантує, що зміна відбудеться тільки якщо жоден інший потік не модифікував значення з моменту його читання.

```
function CAS(location: pointer, expected: value, new_value: value): b
oolean
    atomically:
        if *location == expected then
            *location = new_value
            return true
        else
            return false
```

Операція Push

Операція Push додає новий елемент на вершину стеку. Алгоритм створює новий вузол, встановлює його вказівник next на поточну вершину стеку, а потім намагається атомарно оновити вказівник Top. Якщо інший потік змінив Top під час виконання операції, CAS поверне false, і операція повториться.

```
function Push(stack: pointer to Stack, value: T)
    new_node = new Node()
    new_node.value = value

    while true do
        top = stack.top
        new_node.next = top
```

```
        if CAS(&stack.top, top, new_node) then
            return
```

Цикл while гарантує, що операція буде повторюватися доти, доки не завершиться успішно. На кожній ітерації алгоритм зчитує поточне значення Top і намагається замінити його на новий вузол. Якщо між читанням Top і виконанням CAS інший потік модифікував стек, операція CAS поверне false, і спроба повториться з оновленим значенням Top.

Операція Pop

Операція Pop видаляє та повертає елемент з вершини стеку. Алгоритм зчитує поточну вершину, перевіряє чи стек не порожній, зберігає значення для повернення, а потім намагається атомарно оновити Top на наступний елемент.

```
function Pop(stack: pointer to Stack): T or null
    while true do
        top = stack.top

        if top == null then
            return null  // стек порожній

        next = top.next

        if CAS(&stack.top, top, next) then
            value = top.value
            delete top  // звільнення пам'яті
            return value
```

Важливо відзначити, що значення зчитується до виконання CAS, оскільки після успішного CAS інший потік може видалити вузол. Також алгоритм обробляє випадок порожнього стеку, повертаючи спеціальне значення null.

Проблема ABA

Однією з потенційних проблем стеку Трейбера є так звана ABA-проблема. Вона виникає, коли вказівник змінюється з значення A на B, а потім знову на A між моментом читання та виконання CAS. У цьому випадку CAS успішно виконається, хоча стан структури даних міг суттєво змінитися.

Розглянемо сценарій ABA-проблеми:

1. Потік 1 читає top = A, next = B
2. Потік 2 виконує Pop (видаляє A), потім Pop (видаляє B)
3. Потік 2 виконує Push(A), повертаючи A на вершину

4. Потік 1 виконує CAS(&top, A, B) - успішно, але B вже видалено!

Вирішення ABA-проблеми

Існує кілька підходів до вирішення ABA-проблеми:

- Використання лічильників: Кожен вказівник супроводжується лічильником, який збільшується при кожній зміні. CAS порівнює як вказівник, так і лічильник.

- Hazard pointers: Механізм, що дозволяє потокам “захищати” вузли від видалення іншими потоками.

- Epoch-based reclamation: Відкладене звільнення пам’яті на основі глобальних епох.

Модифікований алгоритм з лічильниками

Для вирішення ABA-проблеми можна використовувати версійовані вказівники:

```
struct VersionedPointer {
    ptr: pointer to Node
    version: integer
}

struct Stack {
    top: VersionedPointer
}

function Push(stack: pointer to Stack, value: T)
    new_node = new Node()
    new_node.value = value

    while true do
        old_top = stack.top
        new_node.next = old_top.ptr

        new_top = VersionedPointer(new_node, old_top.version + 1)

        if CAS(&stack.top, old_top, new_top) then
            return

function Pop(stack: pointer to Stack): T or null
    while true do
        old_top = stack.top

        if old_top.ptr == null then
```

```
            return null

        next = old_top.ptr.next
        new_top = VersionedPointer(next, old_top.version + 1)

        if CAS(&stack.top, old_top, new_top) then
            value = old_top.ptr.value
            // відкладене видалення old_top.ptr
            return value
```

Ініціалізація стеку

Ініціалізація стеку Трейбера дуже проста - достатньо встановити вказівник Top в null:

```
function InitStack(): Stack
    stack = new Stack()
    stack.top = null
    return stack
```

Додаткові операції

Крім базових операцій Push та Pop, можна реалізувати додаткові операції:

```
function IsEmpty(stack: pointer to Stack): boolean
    return stack.top == null

function Peek(stack: pointer to Stack): T or null
    top = stack.top
    if top == null then
        return null
    else
        return top.value
```

Операція Peek повертає значення верхнього елемента без його видалення. Важливо розуміти, що в багатопотоковому середовищі результат Peek може стати неактуальним одразу після повернення.

Область застосування

Стек Трейбера знаходить широке застосування в системах, де потрібна висока продуктивність та масштабованість:

- Системи управління пам’яттю: Для реалізації пулів об’єктів та алокаторів пам’яті без блокувань.

- Планувальники задач: У системах з work-stealing для зберігання локальних черг задач.

- Мережеві стеки: Для буферизації пакетів у високопродуктивних мережевих додатках.

- Системи обробки подій: Як буфер для асинхронних подій у реактивних системах.

- Компілятори та інтерпретатори: Для реалізації стеків викликів у багатопотокових середовищах виконання.

- Ігрові движки: Для управління тимчасовими об’єктами та ресурсами без блокувань.

Приклади застосування

Приклад 1: Базова демонстрація

Розглянемо послідовність операцій зі стеком, що виконуються двома потоками:

Початковий стан: порожній стек

```
Top -> null
```

Потік 1 виконує Push(A):

1. Створює вузол з значенням A
2. Встановлює A.next = null
3. CAS(&Top, null, A) - успішно

Стан після операції:

```
Top -> [A] -> null
```

Потік 2 виконує Push(B) одночасно з тим, як Потік 1 виконує Push(C):

Потік 2:

1. Створює вузол B
2. Читає top = A
3. Встановлює B.next = A
4. Намагається CAS(&Top, A, B)

Потік 1:

1. Створює вузол C
2. Читає top = A
3. Встановлює C.next = A
4. Намагається CAS(&Top, A, C)

Припустимо, Потік 1 виконує CAS першим:

```
Top -> [C] -> [A] -> null
```

Потік 2 виявляє, що CAS не вдався (Top != A), повторює:

1. Читає top = C
2. Встановлює B.next = C
3. CAS(&Top, C, B) - успішно

Фінальний стан:

```
Top -> [B] -> [C] -> [A] -> null
```

Приклад 2: Одночасні операції Pop

Розглянемо стек з трьома елементами:

```
Top -> [X] -> [Y] -> [Z] -> null
```

Два потоки одночасно виконують Pop():

Потік 1: 1. Читає top = X, next = Y 2. Намагається CAS(&Top, X, Y)

Потік 2: 1. Читає top = X, next = Y 2. Намагається CAS(&Top, X, Y)

Тільки один з потоків успішно виконає CAS. Припустимо, це Потік 1:

```
Top -> [Y] -> [Z] -> null
```

Потік 1 повертає X.

Потік 2 виявляє невдачу CAS і повторює:

1. Читає top = Y, next = Z
2. CAS(&Top, Y, Z) - успішно

```
Top -> [Z] -> null
```

Потік 2 повертає Y.

Приклад 3: Демонстрація ABA-проблеми

Початковий стан:

```
Top -> [A] -> [B] -> [C] -> null
```

Послідовність подій:

1. Потік 1 читає top = A, next = B
2. Потік 1 призупиняється
3. Потік 2 виконує Pop() - видаляє A
4. Потік 2 виконує Pop() - видаляє B
5. Потік 2 виконує Push(D)
6. Потік 2 виконує Push(A) - повертає A на вершину

Стан після дій Потоку 2:

```
Top -> [A] -> [D] -> [C] -> null
```

7. Потік 1 відновлюється і виконує CAS(&Top, A, B)

8. CAS успішний, оскільки Top == A

Результат:

```
Top -> [B] -> ??? (B був видалений!)
```

Це призводить до некоректного стану стеку.

Переваги та недоліки

Переваги

- Простота реалізації: Стек Трейбера є одним з найпростіших lock-free алгоритмів.

- Висока продуктивність: Відсутність блокувань забезпечує високу пропускну здатність.

- Відсутність deadlock: Неможливі взаємні блокування між потоками.

- Масштабованість: Продуктивність добре масштабується зі збільшенням кількості потоків.

- Lock-free гарантія: Принаймні один потік завжди зробить прогрес.

- Низька латентність: Операції мають передбачуваний час виконання.

Недоліки

- ABA-проблема: Вимагає додаткових механізмів для надійного вирішення.

- Складність управління пам’яттю: Безпечне звільнення пам’яті в багатопотоковому середовищі є нетривіальною задачею.

- Можливість livelock: При високій конкуренції потоки можуть довго повторювати спроби.

- Обмежена функціональність: Підтримує тільки операції LIFO, на відміну від більш складних структур.

- Залежність від апаратури: Вимагає апаратної підтримки атомарних операцій.

- Складність налагодження: Помилки важко відтворити та виправити.

Контрольні запитання

1. Що таке lock-free структура даних і які її основні властивості?

2. Поясніть роль операції CAS в алгоритмі стеку Трейбера.

3. Чому операції Push та Pop використовують цикли?

4. Що таке ABA-проблема та як вона може виникнути в стеку Трейбера?

5. Які існують методи вирішення ABA-проблеми?

6. Чому значення вузла зчитується до виконання CAS в операції Pop?

7. Які гарантії прогресу надає стек Трейбера?

8. Як забезпечується атомарність операцій в алгоритмі?

9. Чому стек Трейбера масштабується краще за стек з мьютексами?

10. Які обмеження має стек Трейбера порівняно з традиційними реалізаціями?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та аналіз lock-free стеку Трейбера

2. Мета

Вивчити принципи роботи lock-free алгоритмів, освоїти використання атомарних операцій для реалізації неблокуючих структур даних, реалізувати стек Трейбера та дослідити його характеристики в багатопотоковому середовищі.

3. Завдання

Реалізувати стек Трейбера з операціями Push та Pop, створити механізм для демонстрації ABA-проблеми, реалізувати версію з вирішенням ABA-проблеми, провести порівняльний аналіз продуктивності з традиційною реалізацією стеку на основі мьютексів.

4. Опис

Лабораторна робота передбачає поетапну реалізацію стеку Трейбера, починаючи з базової версії та закінчуючи оптимізованою версією з вирішенням ABA-проблеми. Студенти повинні реалізувати тестове середовище для демонстрації коректності роботи в багатопотоковому середовищі та виміряти продуктивність різних реалізацій.

5. Порядок виконання

- Реалізувати структури Node та Stack

- Створити базову версію операцій Push та Pop

- Розробити тестову програму для демонстрації ABA-проблеми

- Реалізувати версію з лічильниками для вирішення ABA-проблеми

- Створити альтернативну реалізацію стеку з використанням мьютексів

- Розробити набір тестів для перевірки коректності

- Провести вимірювання продуктивності для різних сценаріїв

- Побудувати графіки та проаналізувати результати

6. Результат

Результатом виконання роботи є працююча реалізація lock-free стеку Трейбера з вирішенням ABA-проблеми, набір тестів для перевірки коректності, демонстрація ABA-проблеми, порівняльний аналіз продуктивності різних реалізацій, висновки щодо ефективності lock-free підходу.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Реалізація атомарної операції CAS

- Базова версія операцій Push та Pop

- Версія з вирішенням ABA-проблеми

- Реалізація стеку на мьютексах для порівняння

- Тестові сценарії

2. Продемонструвати роботу додатку:

- Коректна робота базових операцій

- Демонстрація ABA-проблеми

- Робота версії з лічильниками

- Результати стрес-тестування

- Графіки продуктивності

3. Бути готовим відповісти на питання:

- Як працює операція CAS?

- Чому виникає ABA-проблема?

- Як лічильники вирішують ABA-проблему?

- Які переваги дає lock-free підхід?

Тести

1. Основна властивість lock-free алгоритму:

- 1) Всі операції виконуються миттєво

- 2) Принаймні один потік завжди зробить прогрес

- 3) Використовується глобальне блокування

- 4) Операції ніколи не повторюються

2. Операція CAS повертає true коли:

- 1) Значення було успішно прочитано

- 2) Операція була скасована

- 3) Поточне значення дорівнює очікуваному і було замінено

- 4) Стек порожній

3. ABA-проблема виникає коли:

- 1) Стек переповнюється

- 2) Два потоки одночасно виконують Push

- 3) Вказівник змінюється і повертається до початкового значення

- 4) Операція Pop виконується на порожньому стеку

4. Чому в операції Pop значення зчитується до CAS?

- 1) Для оптимізації швидкості

- 2) Вузол може бути видалений після CAS

- 3) Це помилка в алгоритмі

- 4) Для уникнення ABA-проблеми

5. Стек Трейбера реалізується як:

- 1) Масив фіксованого розміру

- 2) Двозв’язний список

- 3) Однозв’язний список

- 4) Дерево

Відповіді:

1-2,

2-3,

3-3,

4-2,

5-3

Тести підвищеної складності

1. Яка мінімальна кількість CAS операцій потрібна для успішного Push в найкращому випадку?

- 1) 0

- 2) 1

- 3) 2

- 4) 3

2. Що станеться якщо два потоки одночасно виконають успішний CAS в операції Push?

- 1) Це неможливо через атомарність CAS

- 2) Стек стане некоректним

- 3) Один елемент буде втрачено

- 4) Виникне deadlock

3. Яка часова складність операцій Push та Pop в найгіршому випадку?

- 1) O(1)

- 2) O(log n)

- 3) O(n)

- 4) Необмежена

4. Чому версійовані вказівники вирішують ABA-проблему?

- 1) Вони блокують доступ до вузлів

- 2) Версія змінюється при кожній модифікації

- 3) Вони прискорюють операції

- 4) Вони зменшують використання пам’яті

5. Який недолік має використання 64-бітних лічильників для версій?

- 1) Вони можуть переповнитися

- 2) Не всі архітектури підтримують 128-бітні CAS

- 3) Збільшується складність коду

- 4) Зменшується продуктивність на 50%

6. Коли операція Peek може повернути некоректне значення?

- 1) Ніколи

- 2) Коли стек порожній

- 3) Значення коректне в момент читання, але може змінитися

- 4) При ABA-проблемі

7. Яка основна перевага hazard pointers над лічильниками?

- 1) Простіша реалізація

- 2) Вирішують проблему управління пам’яттю

- 3) Швидші операції

- 4) Менше використання пам’яті

8. Чому стек Трейбера не підходить для реалізації черги?

- 1) Занадто повільний

- 2) Потрібен доступ до обох кінців списку

- 3) ABA-проблема

- 4) Обмеження CAS

9. Скільки разів може повторитися цикл в операції Push при n потоках?

- 1) Максимум n разів

- 2) Максимум 2n разів

- 3) Теоретично необмежено

- 4) Рівно один раз

10. Що є точкою лінеаризації для операції Push?

- 1) Створення нового вузла

- 2) Успішне виконання CAS

- 3) Читання значення top

- 4) Встановлення next вказівника

Відповіді:

1. 2) Потрібна тільки одна успішна CAS операція для оновлення top

2. 1) CAS атомарна, тому тільки один потік може успішно її виконати

3. 4) При постійній конкуренції цикл може повторюватися необмежено

4. 2) Кожна зміна збільшує версію, унеможливлюючи повторення старої пари вказівник-версія

5. 2) Для атомарного оновлення пари вказівник-версія потрібен 128-бітний CAS

6. 3) Між читанням значення та його використанням стек може змінитися

7. 2) Hazard pointers дозволяють безпечно звільняти пам’ять в lock-free алгоритмах

8. 2) Для ефективної черги потрібні операції на обох кінцях, що ускладнює lock-free реалізацію

9. 3) Теоретично при постійних невдачах CAS цикл може тривати необмежено

10. 2) Момент успішного CAS є точкою, коли операція логічно завершується

Завдання для самостійної роботи

1. Реалізуйте версію стеку Трейбера з обмеженням розміру без використання додаткових блокувань.

2. Створіть візуалізацію роботи стеку Трейбера для навчальних цілей з покроковою демонстрацією.

3. Реалізуйте механізм hazard pointers для безпечного управління пам’яттю.

4. Розробіть версію з підтримкою операції MultiPop для видалення кількох елементів.

5. Дослідіть вплив різних стратегій backoff на продуктивність при високій конкуренції.

6. Реалізуйте elimination backoff stack для покращення масштабованості.

7. Створіть гібридну версію, яка перемикається між lock-free та blocking режимами.

8. Розробіть інструмент для автоматичного виявлення ABA-проблем у коді.

9. Реалізуйте wait-free версію стеку та порівняйте з lock-free версією.

10. Дослідіть можливість використання hardware transactional memory для спрощення реалізації.

Теми лабораторних робіт

1. Базова реалізація стеку Трейбера та демонстрація його роботи.

2. Дослідження ABA-проблеми та методів її вирішення.

3. Реалізація пулу об’єктів на основі стеку Трейбера.

4. Розробка бенчмарків для порівняння lock-free та lock-based структур.

5. Створення відмовостійкої версії стеку для розподілених систем.

6. Оптимізація стеку Трейбера для NUMA архітектур.

7. Реалізація складених lock-free структур на основі стеку Трейбера.

8. Інтеграція стеку Трейбера в реальний додаток та аналіз продуктивності.

9. Розробка інструментів профілювання для lock-free структур даних.

10. Створення навчального симулятора для вивчення lock-free алгоритмів.

13.3. Розподілені алгоритми

Алгоритми консенсусу в розподілених системах

Консенсус - це фундаментальна проблема в розподілених системах, яка полягає в досягненні згоди між кількома незалежними вузлами щодо єдиного значення або стану системи. У розподіленому середовищі, де вузли можуть відмовляти, а повідомлення можуть губитися або затримуватися, досягнення такої згоди стає нетривіальною задачею.

Алгоритми консенсусу вирішують критичні проблеми розподілених систем:

1. Забезпечення узгодженості даних

У системах з репліками даних всі копії повинні залишатися синхронізованими. Без консенсусу різні репліки можуть мати різні версії даних, що призведе до некоректної роботи системи.

2. Координація дій

Коли кілька вузлів повинні виконати скоординовані дії (наприклад, розподілена транзакція), необхідно досягти згоди про те, чи виконувати операцію чи скасувати її.

3. Вибір лідера

У багатьох розподілених системах потрібен єдиний координатор або лідер. Алгоритми консенсусу забезпечують вибір одного лідера та запобігають ситуації “split-brain”, коли система має кілька лідерів одночасно.

4. Відмовостійкість

Консенсус дозволяє системі продовжувати працювати навіть при відмові частини вузлів, забезпечуючи високу доступність сервісів.

Алгоритми консенсусу повинні гарантувати три ключові властивості:

- **Узгодженість (Agreement)**: Всі коректно працюючі вузли приймають одне й те саме значення

- **Цілісність (Integrity)**: Прийняте значення було запропоноване одним з вузлів

- **Завершеність (Termination)**: Всі коректно працюючі вузли врешті-решт приймуть рішення

Фішер, Лінч та Патерсон довели, що в повністю асинхронній системі неможливо гарантувати консенсус при відмові навіть одного вузла. Це означає, що практичні алгоритми консенсусу роблять певні припущення про синхронність системи.

Теорема CAP

Система не може одночасно гарантувати консистентність, доступність та стійкість до розділення мережі. Алгоритми консенсусу зазвичай жертвують доступністю заради консистентності.

Популярні алгоритми консенсусу

Paxos

- Перший практично застосовний алгоритм консенсусу

- Математично доведена коректність

- Складний для розуміння та реалізації

- Використовується в Google Chubby, Apache ZooKeeper

Raft

- Створений як простіша альтернатива Paxos

- Чітке розділення на підпроблеми: вибір лідера, реплікація логу, безпека

- Широко використовується: etcd, Consul, CockroachDB

- Легший для розуміння та реалізації

PBFT (Practical Byzantine Fault Tolerance)

- Толерує Byzantine failures (зловмисну поведінку вузлів)

- Використовується в блокчейн системах

- Вища складність порівняно з Paxos/Raft

Proof of Work / Proof of Stake

- Використовуються в публічних блокчейнах

- Забезпечують консенсус в повністю децентралізованих системах

- Високі витрати ресурсів (PoW) або складність реалізації (PoS)

Алгоритми консенсусу є основою для:

- Розподілених баз даних: MongoDB, CockroachDB, Amazon DynamoDB

- Систем координації: Apache ZooKeeper, etcd, Consul

- Блокчейн платформ: Bitcoin, Ethereum, Hyperledger

- Хмарних сервісів: системи управління контейнерами, розподілені файлові системи

- Фінансових систем: забезпечення консистентності транзакцій

Консенсус Paxos

Теоретичні відомості

Алгоритм Paxos, розроблений Леслі Лемпортом у 1989 році та опублікований у 1998 році, є фундаментальним алгоритмом досягнення консенсусу в розподілених системах. Він вирішує одну з найскладніших проблем розподілених обчислень - як досягти згоди між кількома вузлами щодо єдиного значення в умовах можливих відмов окремих компонентів системи.

Проблема консенсусу

Проблема консенсусу полягає в тому, що група процесів або вузлів у розподіленій системі повинна узгодити спільне значення. Ця проблема ускладнюється тим, що вузли можуть відмовляти, повідомлення можуть губитися або затримуватися, а мережа може розділятися на ізольовані частини. Незважаючи на ці складнощі, система повинна гарантувати три ключові властивості:

- Узгодженість (Agreement): Всі коректно працюючі вузли повинні прийняти одне й те саме значення.

- Цілісність (Integrity): Прийняте значення повинно бути запропоноване одним з вузлів.

- Завершеність (Termination): Всі коректно працюючі вузли врешті-решт приймуть якесь значення.

Основні компоненти Paxos

Алгоритм Paxos визначає три ролі, які можуть виконувати вузли системи:

- Proposer: Вузол, який пропонує значення для консенсусу. Proposer ініціює процес досягнення згоди, відправляючи пропозиції іншим вузлам.

- Acceptor: Вузол, який голосує за пропозиції. Acceptor зберігає інформацію про попередні голосування та гарантує, що буде прийнято лише одне значення.

- Learner: Вузол, який дізнається про прийняте значення. Learner не бере участі в процесі голосування, але отримує інформацію про результат консенсусу.

Важливо зазначити, що один фізичний вузол може виконувати кілька ролей одночасно. Наприклад, вузол може бути одночасно Proposer та Acceptor.

Основний алгоритм Paxos

Алгоритм Paxos складається з двох фаз: фази підготовки та фази прийняття. Кожна пропозиція ідентифікується унікальним номером, який складається з лічильника та ідентифікатора вузла. Номери пропозицій повинні бути унікальними та впорядкованими.

Фаза 1: Підготовка

На цій фазі Proposer намагається зарезервувати номер пропозиції та дізнатися про попередні голосування:

```
function Proposer_Phase1(proposer_id, value_to_propose)
    // Генеруємо унікальний номер пропозиції
    proposal_number = generate_proposal_number(proposer_id)

    // Відправляємо запит Prepare всім Acceptor'ам
    for each acceptor in acceptors do
        send_prepare_request(acceptor, proposal_number)

    // Чекаємо відповіді від більшості Acceptor'ів
    responses = wait_for_majority_responses()

    if received_majority_responses(responses) then
        // Аналізуємо відповіді
        highest_numbered_proposal = find_highest_numbered_accepted_pr
oposal(responses)

        if highest_numbered_proposal exists then
            // Використовуємо значення з попередньої пропозиції
            value_to_propose = highest_numbered_proposal.value
```

```
        // Переходимо до фази 2
        return Phase2(proposal_number, value_to_propose)
    else
        // Не вдалося отримати більшість, повторюємо з новим номером
        return retry_with_higher_number()
```

Коли Acceptor отримує запит Prepare, він виконує наступні дії:

```
function Acceptor_Handle_Prepare(proposal_number)
    if proposal_number > promised_proposal_number then
        // Обіцяємо не приймати пропозиції з меншими номерами
        promised_proposal_number = proposal_number

        // Відправляємо відповідь з інформацією про прийняті пропозиц
ії
        if accepted_proposal exists then
            send_promise(proposal_number, accepted_proposal_number, a
ccepted_value)
        else
            send_promise(proposal_number, null, null)
    else
        // Відхиляємо запит, оскільки вже обіцяли вищий номер
        send_reject(promised_proposal_number)
```

Фаза 2: Прийняття

На цій фазі Proposer намагається отримати прийняття свого значення:

```
function Proposer_Phase2(proposal_number, value)
    // Відправляємо запит Accept всім Acceptor'ам
    for each acceptor in acceptors do
        send_accept_request(acceptor, proposal_number, value)

    // Чекаємо відповіді від більшості
    responses = wait_for_majority_responses()

    if received_majority_accepts(responses) then
        // Значення прийнято!
        // Повідомляємо Learner'ів
        notify_learners(proposal_number, value)
        return SUCCESS
    else
        // Не вдалося, потрібно почати спочатку
        return FAILED
```

Обробка запиту Accept на стороні Acceptor:

```
function Acceptor_Handle_Accept(proposal_number, value)
    if proposal_number >= promised_proposal_number then
```

```
        // Приймаємо пропозицію
        promised_proposal_number = proposal_number
        accepted_proposal_number = proposal_number
        accepted_value = value

        // Зберігаємо стан на диск для відмовостійкості
        persist_state(promised_proposal_number, accepted_proposal_num
ber, accepted_value)

        send_accepted(proposal_number)
    else
        // Відхиляємо, оскільки обіцяли вищий номер
        send_reject(promised_proposal_number)
```

Multi-Paxos

Basic Paxos вирішує проблему досягнення консенсусу для одного значення. Однак у реальних системах часто потрібно узгоджувати послідовність значень. Multi-Paxos є оптимізацією, яка дозволяє ефективно узгоджувати послідовність значень, пропускаючи фазу Prepare для наступних значень, якщо лідер не змінився.

```
function MultiPaxos_Leader(values_to_propose)
    // Виконуємо фазу 1 один раз
    proposal_number = Phase1_for_all_slots()

    // Для кожного значення виконуємо тільки фазу 2
    for each value in values_to_propose do
        slot = get_next_slot()
        Phase2(proposal_number, slot, value)
```

Обробка відмов та відновлення

Paxos спроектований для роботи в умовах відмов. Розглянемо основні сценарії:

Відмова Proposer

Якщо Proposer відмовляє під час виконання алгоритму, інший Proposer може почати новий раунд з вищим номером пропозиції. Алгоритм гарантує, що навіть якщо попередній Proposer частково завершив свою роботу, новий Proposer дізнається про це та продовжить з правильним значенням.

Відмова Acceptor

Система може толерувати відмову до `(n-1)/2` Acceptor'ів, де `n` - загальна кількість Acceptor'ів. Це випливає з вимоги отримання відповідей від більшості для прогресу алгоритму.

Розділення мережі

При розділенні мережі частина системи з більшістю Acceptor'ів може продовжувати роботу, тоді як менша частина буде заблокована до відновлення зв'язку.

Гарантії безпеки та живучості

Paxos надає сильні гарантії безпеки:

- Унікальність: Для кожного слоту може бути прийнято лише одне значення.

- Стабільність: Після прийняття значення воно ніколи не змінюється.

- Коректність: Прийняте значення завжди було запропоноване якимось Proposer.

Щодо живучості, Paxos гарантує прогрес за умови:

- Більшість Acceptor'ів працюють коректно

- Мережа врешті-решт доставляє повідомлення

- Існує період стабільності для завершення протоколу

Область застосування

Алгоритм Paxos широко використовується в критично важливих розподілених системах:

- Системи управління конфігурацією: Google Chubby, Apache ZooKeeper використовують варіанти Paxos для зберігання конфігураційної інформації.

- Розподілені бази даних: Системи як Google Spanner, Amazon DynamoDB використовують Paxos для репликації даних та забезпечення консистентності.

- Координація в хмарних системах: Для вибору лідера, розподілу ресурсів та координації між сервісами.

- Блокчейн системи: Деякі блокчейн платформи використовують Paxos як альтернативу proof-of-work для досягнення консенсусу.

- Відмовостійкі системи зберігання: Для забезпечення консистентності реплік даних у розподілених сховищах.

Приклади застосування

Приклад 1: Базовий сценарій успішного консенсусу

Розглянемо систему з трьома вузлами (A, B, C), які виконують ролі і Proposer, і Acceptor. Вузол A хоче запропонувати значення “X”.

Початковий стан:

- Всі Acceptor’и мають promised = 0, accepted = null

Фаза 1 (Prepare):

1. A генерує proposal_number = 1.1 (1 - лічильник, 1 - ID вузла A)
2. A відправляє Prepare(1.1) до A, B, C
3. Acceptor’и отримують Prepare(1.1):

- A: promised = 1.1, відповідає Promise(1.1, null)
- B: promised = 1.1, відповідає Promise(1.1, null)
- C: promised = 1.1, відповідає Promise(1.1, null)

4. A отримує 3 Promise (більшість), жодних попередніх значень

Фаза 2 (Accept):

1. A відправляє Accept(1.1, “X”) до A, B, C
2. Acceptor’и отримують Accept(1.1, “X”):

- A: accepted = (1.1, “X”), відповідає Accepted(1.1)
- B: accepted = (1.1, “X”), відповідає Accepted(1.1)
- C: accepted = (1.1, “X”), відповідає Accepted(1.1)

3. A отримує 3 Accepted - консенсус досягнуто для значення “X”

Приклад 2: Конфлікт пропозицій

Два вузли A та B одночасно намагаються запропонувати різні значення.

Початковий стан:

- Всі Acceptor’и мають promised = 0, accepted = null
- A хоче запропонувати “X”
- B хоче запропонувати “Y”

Паралельна фаза 1:

1. A відправляє Prepare(1.1)
2. B відправляє Prepare(1.2)
3. Acceptor C отримує спочатку Prepare(1.1), потім Prepare(1.2):

- Для Prepare(1.1): promised = 1.1, Promise(1.1, null)
- Для Prepare(1.2): promised = 1.2, Promise(1.2, null)

4. Acceptor’и A та B отримують обидва запити в різному порядку

Результат фази 1:

- A отримує більшість Promise для 1.1 від себе та ще одного вузла
- B отримує більшість Promise для 1.2 від себе та C

Фаза 2:

1. A відправляє Accept(1.1, “X”)
2. B відправляє Accept(1.2, “Y”)
3. Acceptor C відхиляє Accept(1.1, “X”), бо promised = 1.2 > 1.1
4. A не отримує більшість, починає новий раунд

Новий раунд A:

1. A відправляє Prepare(2.1)
2. Отримує Promise з інформацією про прийняте значення “Y”
3. A змушений запропонувати “Y” у фазі Accept
4. Консенсус досягнуто для значення “Y”

Приклад 3: Відновлення після часткової відмови

Розглянемо сценарій, де Proposer відмовляє посеред виконання алгоритму.

Ситуація:

- 5 Acceptor’ів (A, B, C, D, E)
- Proposer P1 відправив Accept(1.1, “X”) до A, B, C
- A та B прийняли, C ще обробляє
- P1 відмовляє

Відновлення:

1. Новий Proposer P2 починає з Prepare(2.1)
2. Відповіді:
    - A: Promise(2.1, accepted=(1.1, “X”))
    - B: Promise(2.1, accepted=(1.1, “X”))
    - C: Promise(2.1, null)
    - D: Promise(2.1, null)

- E: Promise(2.1, null)

3. P2 бачить, що було частково прийнято “X”

4. P2 відправляє Accept(2.1, “X”)

5. Всі Acceptor’и приймають - консенсус завершено

Переваги та недоліки

Переваги

- Доведена коректність: Математично доведено, що алгоритм гарантує безпеку консенсусу.

- Відмовостійкість: Толерує відмову до половини вузлів без втрати працездатності.

- Відсутність єдиної точки відмови: Не вимагає постійного лідера або центрального координатора.

- Гнучкість ролей: Вузли можуть виконувати кілька ролей, що спрощує розгортання.

- Підтримка довільних значень: Може узгоджувати будь-які значення, не лише бінарні.

Недоліки

- Складність розуміння: Алгоритм важкий для розуміння та правильної реалізації.

- Можливість блокування: За певних умов система може увійти в стан, коли прогрес неможливий.

- Накладні витрати: Вимагає багато повідомлень для досягнення консенсусу.

- Відсутність гарантій живучості: Не гарантує завершення за наявності постійних відмов.

- Складність налагодження: Помилки в розподілених системах на основі Paxos важко відтворити та виправити.

Контрольні запитання

1. Які три основні ролі визначає алгоритм Paxos та їх функції?

2. Чому в алгоритмі Paxos потрібні дві фази для досягнення консенсусу?

3. Як Paxos гарантує, що буде прийнято лише одне значення?

4. Що таке номер пропозиції та чому він повинен бути унікальним?

5. Як алгоритм обробляє ситуацію, коли Proposer дізнається про раніше прийняте значення?

6. Скільки вузлів може відмовити в системі з n Acceptor’ів без втрати працездатності?

7. Що таке Multi-Paxos та яку оптимізацію він надає?

8. Які гарантії безпеки надає алгоритм Paxos?

9. За яких умов Paxos гарантує прогрес системи?

10. Як вирішується конфлікт між двома паралельними пропозиціями?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та аналіз алгоритму консенсусу Paxos

2. Мета

Вивчити принципи досягнення консенсусу в розподілених системах, реалізувати алгоритм Paxos, дослідити його поведінку в різних сценаріях відмов та навчитися проектувати відмовостійкі розподілені системи.

3. Завдання

Реалізувати повнофункціональну систему на основі Paxos, що включає всі три ролі (Proposer, Acceptor, Learner), змоделювати різні сценарії відмов, реалізувати Multi-Paxos для послідовності значень, створити візуалізацію роботи алгоритму.

4. Опис

Лабораторна робота передбачає поетапну реалізацію алгоритму Paxos, починаючи з базової версії для одного значення та розширюючи до Multi-Paxos. Студенти повинні реалізувати механізми обробки відмов, персистентність стану та засоби моніторингу роботи системи.

5. Порядок виконання

- Реалізувати базові структури даних для повідомлень Paxos
- Створити реалізацію ролей Proposer, Acceptor, Learner
- Реалізувати мережевий шар з симуляцією втрат та затримок
- Додати персистентність стану Acceptor’ів
- Реалізувати базовий Paxos для одного значення
- Розширити до Multi-Paxos

- Створити тестові сценарії для різних видів відмов
- Реалізувати візуалізацію обміну повідомленнями
- Провести експерименти з продуктивністю

6. Результат

Працююча реалізація Paxos з можливістю конфігурації топології, демонстрація роботи в різних сценаріях відмов, візуалізація процесу досягнення консенсусу, аналіз продуктивності та масштабованості, документація з описом реалізації.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
- Реалізація фази Prepare для Proposer та Acceptor
- Реалізація фази Accept для Proposer та Acceptor
- Механізм вибору номерів пропозицій
- Логіка обробки відмов та повторних спроб
- Персистентність стану

2. Продемонструвати роботу додатку:
- Успішний консенсус без відмов
- Консенсус при відмові Proposer
- Консенсус при відмові меншості Acceptor’ів
- Конфлікт пропозицій та його вирішення
- Робота Multi-Paxos

3. Бути готовим відповісти на питання:
- Чому потрібні дві фази в алгоритмі?
- Як гарантується унікальність прийнятого значення?
- Як працює механізм відновлення після відмов?
- Які оптимізації можливі для покращення продуктивності?

Тести

1. Мінімальна кількість Acceptor’ів для досягнення консенсусу:
- 1) Один
- 2) Два

- 3) Більшість від загальної кількості
- 4) Всі Acceptor’и

2. Що робить Proposer, якщо дізнається про раніше прийняте значення:
- 1) Відміняє свою пропозицію
- 2) Ігнорує попереднє значення
- 3) Пропонує знайдене значення
- 4) Починає нову фазу Prepare

3. Номер пропозиції складається з:
- 1) Тільки лічильника
- 2) Лічильника та ID вузла
- 3) Часової мітки
- 4) Випадкового числа

4. Multi-Paxos оптимізує:
- 1) Фазу Accept
- 2) Фазу Prepare для послідовних значень
- 3) Кількість Acceptor’ів
- 4) Розмір повідомлень

5. Paxos гарантує прогрес за умови:
- 1) Відсутності відмов
- 2) Наявності одного Proposer
- 3) Більшість Acceptor’ів працюють та є період стабільності
- 4) Всі вузли працюють

Відповіді:

1-3,

2-3,

3-2,

4-2,

5-3

Тести підвищеної складності

1. Чому Acceptor повинен зберігати два значення (promised та accepted)?

- 1) Для оптимізації пам’яті

- 2) Для резервного копіювання

- 3) Promised блокує старі пропозиції, accepted зберігає прийняте значення

- 4) Це помилка в алгоритмі

2. Що станеться, якщо всі Proposer’и використовують однаковий ID вузла?

- 1) Алгоритм працюватиме нормально

- 2) Можливі колізії номерів пропозицій

- 3) Покращиться продуктивність

- 4) Зменшиться кількість повідомлень

3. Яка мінімальна кількість раундів потрібна для досягнення консенсусу без конфліктів?

- 1) 1 раунд (2 фази)

- 2) 2 раунди

- 3) 3 раунди

- 4) Залежить від кількості вузлів

4. Як Paxos обробляє byzantine failures?

- 1) Повністю толерує

- 2) Частково толерує

- 3) Не толерує, розрахований на crash failures

- 4) Вимагає додаткової криптографії

5. Чому в Promise повідомленні передається інформація про accepted значення?

- 1) Для статистики

- 2) Щоб новий Proposer міг дізнатися та продовжити попередній консенсус

- 3) Для оптимізації мережі

- 4) Це необов’язково

6. Що таке “dueling proposers” проблема?

- 1) Два Proposer’и пропонують однакове значення

- 2) Proposer’и постійно перебивають один одного вищими номерами

- 3) Відмова двох Proposer’ів одночасно

- 4) Конфлікт ID вузлів

7. Як FLP теорема пов’язана з Paxos?

- 1) Paxos порушує FLP теорему

- 2) Paxos обходить FLP через часткову синхронність

- 3) FLP не стосується Paxos

- 4) Paxos доводить хибність FLP

8. Яка роль Learner’ів в алгоритмі?

- 1) Вони голосують за пропозиції

- 2) Вони тільки дізнаються про прийняті значення

- 3) Вони генерують пропозиції

- 4) Вони зберігають стан

9. Скільки повідомлень потрібно для Basic Paxos у найкращому випадку?

- 1) O(n)

- 2) $O(n^2)$

- 3) O(log n)

- 4) O(1)

10. Чому Paxos називають “синхронним” алгоритмом у асинхронній моделі?

- 1) Він вимагає синхронізованих годинників

- 2) Він вимагає періодів стабільності для прогресу

- 3) Всі операції синхронні

- 4) Це помилкове твердження

Відповіді:

1. 3) Promised запобігає прийняттю старих пропозицій, accepted зберігає фактично прийняте значення

2. 2) Без унікальних ID можуть виникати однакові номери пропозицій у різних Proposer’ів

3. 1) За відсутності конфліктів достатньо одного раунду з двох фаз

4. 3) Basic Paxos розрахований тільки на crash failures, не на byzantine

5. 2) Це ключовий механізм збереження раніше прийнятих значень

6. 2) Проблема виникає коли Proposer’и постійно підвищують номери, блокуючи один одного

7. 2) FLP забороняє детерміністичний консенсус в асинхронній моделі, Paxos використовує часткову синхронність

8. 2) Learner’и не беруть участі в голосуванні, лише отримують результат

9. 2) Потрібно O(n) повідомлень для кожної фази, всього O(n) для двох фаз

10. 2) Для гарантії прогресу потрібні періоди без збоїв та затримок

Завдання для самостійної роботи

1. Реалізуйте Byzantine Paxos, який толерує Byzantine failures використовуючи криптографічні підписи.

2. Створіть оптимізовану версію Fast Paxos, яка дозволяє досягти консенсусу за одну фазу в оптимальних умовах.

3. Розробіть механізм динамічної реконфігурації складу Acceptor’ів без зупинки системи.

4. Реалізуйте Cheap Paxos з використанням додаткових вузлів тільки при відмовах.

5. Створіть гібридну систему, що автоматично перемикається між Paxos та Raft залежно від умов.

6. Дослідіть та реалізуйте різні стратегії вибору номерів пропозицій для мінімізації конфліктів.

7. Розробіть систему моніторингу та профілювання для виявлення вузьких місць в Paxos.

8. Реалізуйте версію з підтримкою географічно розподілених датацентрів.

9. Створіть фреймворк для автоматичного тестування коректності реалізацій Paxos.

10. Дослідіть можливості використання машинного навчання для оптимізації параметрів Paxos.

Теми лабораторних робіт

1. Базова реалізація Paxos та візуалізація процесу консенсусу.

2. Порівняльний аналіз Paxos та Raft в різних сценаріях відмов.

3. Реалізація розподіленого key-value сховища на основі Multi-Paxos.

4. Створення системи розподілених блокувань використовуючи Paxos.

5. Розробка відмовостійкої системи конфігурації для мікросервісів.

6. Реалізація Flexible Paxos з різними quorum для читання та запису.

7. Інтеграція Paxos в систему оркестрації контейнерів.

8. Створення розподіленої черги повідомлень з гарантіями Paxos.

9. Реалізація системи вибору лідера для розподілених обчислень.

10. Розробка навчального симулятора для вивчення алгоритмів консенсусу.

Консенсус Raft

Теоретичні відомості

Алгоритм Raft, розроблений Дієго Онгаро та Джоном Оустерхаутом у 2014 році, є алгоритмом досягнення консенсусу в розподілених системах, створеним як більш зрозуміла альтернатива алгоритму Paxos. Головною метою при розробці Raft було створення алгоритму, який би забезпечував ті самі гарантії безпеки та живучості, що й Paxos, але був би значно простішим для розуміння та реалізації.

Основні принципи Raft

Raft розділяє проблему консенсусу на три відносно незалежні підпроблеми:

- Вибори лідера (Leader election): Система повинна вибрати одного лідера серед усіх вузлів.

- Реплікація логу (Log replication): Лідер приймає команди від клієнтів та реплікує їх на інші вузли.

- Безпека (Safety): Гарантії того, що всі вузли приймають однакові рішення в однаковому порядку.

Стани вузлів

У Raft кожен вузол може знаходитися в одному з трьох станів:

- Follower: Пасивний стан, вузол відповідає на запити від лідера та кандидатів.

- Candidate: Проміжний стан під час виборів, вузол намагається стати лідером.

- Leader: Активний стан, вузол обробляє всі клієнтські запити та керує реплікацією.

Всі вузли починають роботу як Follower. Якщо Follower не отримує повідомлень від лідера протягом певного часу (election timeout), він переходить у стан Candidate та ініціює вибори.

Час у Raft поділяється на терміни довільної довжини. Кожен термін починається з виборів, під час яких один або кілька кандидатів намагаються стати лідером. Якщо кандидат виграє вибори, він стає лідером на решту терміну. Терміни нумеруються послідовними цілими числами і служать логічним годинником для системи.

```
struct NodeState {
    currentTerm: integer      // поточний термін вузла
    votedFor: NodeId or null  // за кого голосував у поточному термін
і
    log: array of LogEntry    // журнал команд
    state: {Follower, Candidate, Leader}

    // Індекси для управління логом
    commitIndex: integer      // індекс останньої закоммітеної записи
    lastApplied: integer      // індекс останньої застосованої записи

    // Тільки для лідера
    nextIndex: map[NodeId, integer]   // наступний індекс для кожного
вузла
    matchIndex: map[NodeId, integer]  // відомий збіжний індекс для к
ожного вузла
}

struct LogEntry {
    term: integer    // термін, коли запис був створений
    index: integer   // позиція в логу
    command: any     // команда для state machine
}
```

Вибори лідера

Процес виборів лідера є критичним компонентом Raft. Коли вузол стає кандидатом, він виконує такі дії:

```
function StartElection(node)
    node.state = Candidate
    node.currentTerm = node.currentTerm + 1
    node.votedFor = node.id  // голосує за себе
    votes_received = 1

    // Відправляє RequestVote всім іншим вузлам
```

```
    for each peer in peers do
        send_async RequestVote(
            term = node.currentTerm,
            candidateId = node.id,
            lastLogIndex = length(node.log),
            lastLogTerm = node.log[lastLogIndex].term
        ) to peer

    // Встановлює таймер виборів
    election_timer = random(150ms, 300ms)

    // Чекає на відповіді
    while node.state == Candidate and not election_timer.expired do
        if received VoteResponse from peer then
            if response.term > node.currentTerm then
                node.currentTerm = response.term
                BecomeFollower(node)
                return

            if response.voteGranted then
                votes_received = votes_received + 1
                if votes_received > length(peers) / 2 then
                    BecomeLeader(node)
                    return

    // Таймаут - починаємо нові вибори
    if node.state == Candidate then
        StartElection(node)
```

Обробка запиту на голосування:

```
function HandleRequestVote(node, request)
    response = VoteResponse(term = node.currentTerm, voteGranted = fa
lse)

    // Оновлюємо термін якщо потрібно
    if request.term > node.currentTerm then
        node.currentTerm = request.term
        node.votedFor = null
        BecomeFollower(node)

    // Перевіряємо чи можемо голосувати за кандидата
    if request.term < node.currentTerm then
        return response  // відмова - застарілий термін

    // Перевіряємо обмеження голосування
```

```
    if (node.votedFor == null or node.votedFor == request.candidateId
) and
       IsUpToDate(request.lastLogIndex, request.lastLogTerm, node.log
) then
        node.votedFor = request.candidateId
        response.voteGranted = true
        ResetElectionTimer(node)

    return response

function IsUpToDate(candidateLastIndex, candidateLastTerm, myLog)
    myLastTerm = myLog[length(myLog)].term

    if candidateLastTerm != myLastTerm then
        return candidateLastTerm > myLastTerm

    return candidateLastIndex >= length(myLog)
```

Реплікація логу

Після обрання лідер відповідає за реплікацію логу команд на всі інші вузли. Клієнти надсилають команди лідеру, який додає їх до свого логу та реплікує на followers:

```
function ClientRequest(leader, command)
    // Додаємо нову запис до логу лідера
    entry = LogEntry(
        term = leader.currentTerm,
        index = length(leader.log) + 1,
        command = command
    )
    leader.log.append(entry)

    // Відправляємо AppendEntries всім followers
    for each follower in followers do
        SendAppendEntries(leader, follower)

    // Чекаємо поки запис буде закоммічений
    wait until entry.index <= leader.commitIndex

    // Застосовуємо команду та повертаємо результат клієнту
    result = ApplyCommand(command)
    return result to client

function SendAppendEntries(leader, follower)
    prevLogIndex = leader.nextIndex[follower] - 1
```

```
    prevLogTerm = 0
    if prevLogIndex > 0 then
        prevLogTerm = leader.log[prevLogIndex].term

    entries = leader.log[leader.nextIndex[follower]:]

    send AppendEntries(
        term = leader.currentTerm,
        leaderId = leader.id,
        prevLogIndex = prevLogIndex,
        prevLogTerm = prevLogTerm,
        entries = entries,
        leaderCommit = leader.commitIndex
    ) to follower
```

Обробка AppendEntries на follower:

```
function HandleAppendEntries(node, request)
    response = AppendEntriesResponse(
        term = node.currentTerm,
        success = false
    )

    // Перевірка терміну
    if request.term < node.currentTerm then
        return response

    // Оновлюємо термін та стаємо follower якщо потрібно
    if request.term > node.currentTerm then
        node.currentTerm = request.term
        node.votedFor = null

    BecomeFollower(node)
    ResetElectionTimer(node)

    // Перевірка узгодженості логу
    if request.prevLogIndex > 0 then
        if length(node.log) < request.prevLogIndex or
           node.log[request.prevLogIndex].term != request.prevLogTerm
then
            return response  // неузгоджений лог

    // Видаляємо конфліктуючі записи
    for i = 1 to length(request.entries) do
        index = request.prevLogIndex + i
        if index <= length(node.log) then
```

```
            if node.log[index].term != request.entries[i].term then
                // Видаляємо цей та всі наступні записи
                node.log = node.log[1:index-1]
                break

    // Додаємо нові записи
    for entry in request.entries do
        if entry.index > length(node.log) then
            node.log.append(entry)

    // Оновлюємо commitIndex
    if request.leaderCommit > node.commitIndex then
        node.commitIndex = min(request.leaderCommit, length(node.log)
)
        ApplyCommittedEntries(node)

    response.success = true
    return response
```

Лідер вирішує коли запис може бути закоммічений (застосований до state machine). Запис вважається закомміченим, коли він збережений на більшості вузлів:

```
function UpdateCommitIndex(leader)
    // Знаходимо найбільший індекс N такий, що:
    // 1. Більшість вузлів мають log[N]
    // 2. log[N].term == currentTerm

    for N = length(leader.log) downto leader.commitIndex + 1 do
        if leader.log[N].term == leader.currentTerm then
            replicatedCount = 1  // лідер має цей запис

            for each follower in followers do
                if leader.matchIndex[follower] >= N then
                    replicatedCount = replicatedCount + 1

            if replicatedCount > (length(nodes) / 2) then
                leader.commitIndex = N
                ApplyCommittedEntries(leader)
                break

function ApplyCommittedEntries(node)
    while node.lastApplied < node.commitIndex do
        node.lastApplied = node.lastApplied + 1
        entry = node.log[node.lastApplied]
        ApplyToStateMachine(entry.command)
```

Raft забезпечує кілька важливих гарантій безпеки:

- Election Safety: В кожному терміні може бути обрано максимум одного лідера.

- Leader Append-Only: Лідер ніколи не перезаписує або не видаляє записи зі свого логу.

- Log Matching: Якщо два логи містять запис з однаковим індексом та терміном, то логи ідентичні до цього індексу включно.

- Leader Completeness: Якщо запис закоммічений в певному терміні, то він буде присутній в логах усіх майбутніх лідерів.

- State Machine Safety: Якщо вузол застосував запис з певним індексом до своєї state machine, жоден інший вузол не застосує інший запис з тим самим індексом.

Зміна конфігурації кластера

Raft підтримує динамічну зміну складу кластера без зупинки системи. Для безпечної зміни конфігурації використовується двофазний підхід:

```
function ChangeConfiguration(leader, newConfig)
    // Фаза 1: Перехід до спільної конфігурації
    jointConfig = CreateJointConfig(leader.currentConfig, newConfig)
    jointEntry = LogEntry(
        term = leader.currentTerm,
        command = ConfigChange(jointConfig)
    )

    // Реплікуємо спільну конфігурацію
    ReplicateAndCommit(leader, jointEntry)

    // Фаза 2: Перехід до нової конфігурації
    newEntry = LogEntry(
        term = leader.currentTerm,
        command = ConfigChange(newConfig)
    )

    ReplicateAndCommit(leader, newEntry)
```

Оптимізації та практичні аспекти

Для ефективної роботи Raft використовує кілька оптимізацій:

- Пакетування: Лідер може групувати кілька клієнтських запитів в один AppendEntries RPC.

- Попереднє голосування: Кандидат спочатку перевіряє чи має шанси виграти вибори перед збільшенням терміну.

- Стиснення логу: Періодичне створення знімків стану для видалення старих записів логу.

```
function CreateSnapshot(node)
    snapshot = Snapshot(
        lastIncludedIndex = node.lastApplied,
        lastIncludedTerm = node.log[node.lastApplied].term,
        state = SerializeStateMachine()
    )

    // Зберігаємо знімок
    SaveSnapshotToDisk(snapshot)

    // Видаляємо застарілі записи логу
    node.log = node.log[snapshot.lastIncludedIndex + 1:]

    // Відправляємо знімок відсталим followers
    for each follower in followers do
        if leader.nextIndex[follower] <= snapshot.lastIncludedIndex t
hen
            SendSnapshot(leader, follower, snapshot)
```

Область застосування

Raft широко використовується в сучасних розподілених системах:

- Розподілені бази даних: etcd, CockroachDB, TiDB використовують Raft для забезпечення консистентності даних.

- Системи координації: Consul, etcd служать як розподілені key-value сховища для координації сервісів.

- Розподілені файлові системи: Для забезпечення консистентності метаданих та реплікації.

- Блокчейн платформи: Деякі приватні блокчейни використовують Raft замість proof-of-work.

Переваги та недоліки

Переваги

- Простота розуміння: Чітке розділення на підпроблеми робить алгоритм інтуїтивно зрозумілим.

- Сильний лідер: Спрощує дизайн системи та забезпечує лінеаризованість операцій.

- Детермінованість: Менше недетермінованої поведінки порівняно з Paxos.

- Ефективність: У нормальному режимі потрібен лише один round-trip до більшості вузлів.

- Легкість реалізації: Структурований підхід спрощує правильну реалізацію.

- Практичність: Включає рішення для практичних проблем як зміна конфігурації та компактування логу.

Недоліки

- Залежність від лідера: Відмова лідера блокує систему до обрання нового.

- Чутливість до мережевих розділень: Меншість вузлів не може працювати автономно.

- Накладні витрати heartbeat: Постійний обмін повідомленнями навіть без активності.

- Проблеми масштабування: Один лідер може стати вузьким місцем для великих кластерів.

- Затримки при виборах: Процес виборів може зайняти кілька секунд.

Контрольні запитання

1. Які три основні стани може мати вузол у Raft та їх призначення?

2. Що таке термін у контексті Raft та яку роль він відіграє?

3. Як Raft гарантує, що в одному терміні буде обрано лише одного лідера?

4. Які умови повинен перевірити вузол перед голосуванням за кандидата?

5. Коли лідер вважає запис закомміченим та може застосувати його?

6. Як Raft вирішує проблему неузгоджених логів після зміни лідера?

7. Що таке election timeout та як він впливає на роботу системи?

8. Які п’ять гарантій безпеки надає Raft?

9. Як працює двофазний підхід до зміни конфігурації кластера?

10. Чим відрізняється commitIndex від lastApplied?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму консенсусу Raft

2. Мета

Вивчити принципи роботи алгоритму Raft, реалізувати повнофункціональний Raft-кластер, дослідити поведінку системи в різних сценаріях відмов та навчитися проектувати надійні розподілені системи.

3. Завдання

Реалізувати Raft-кластер з підтримкою виборів лідера, реплікації логу та обробки відмов. Створити клієнтський додаток для тестування. Реалізувати візуалізацію стану кластера. Провести експерименти з різними конфігураціями та сценаріями відмов.

4. Опис

Лабораторна робота передбачає покрокову реалізацію всіх компонентів Raft: механізму виборів, реплікації логу, обробки RPC запитів. Студенти повинні реалізувати персистентність стану, механізм heartbeat, обробку мережевих розділень. Окрема увага приділяється тестуванню та візуалізації.

5. Порядок виконання

- Реалізувати базові структури даних та RPC повідомлення
- Створити state machine для переходів між станами вузла
- Реалізувати механізм виборів лідера
- Додати реплікацію логу та обробку AppendEntries
- Реалізувати персистентність критичного стану
- Створити механізм застосування закоммічених команд
- Додати обробку мережевих затримок та втрат
- Реалізувати візуалізацію стану кластера
- Створити набір автоматизованих тестів
- Провести експерименти з продуктивністю

6. Результат

Функціонуючий Raft-кластер з можливістю додавання/видалення вузлів, візуалізація процесу виборів та реплікації, демонстрація роботи при різних видах відмов, аналіз часу відновлення після відмов, порівняння з іншими алгоритмами консенсусу.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Реалізація RequestVote та AppendEntries RPC
- Механізм часу очікування виборів та сигналу підтвердження
- Логіка переходів між станами
- Алгоритм визначення закоммічених записів
- Персистентність стану на диск

2. Продемонструвати роботу додатку:
- Нормальна робота з реплікацією команд
- Вибори при відмові лідера
- Робота при мережевому розділенні
- Відновлення вузла після відмови
- Зміна конфігурації кластера

3. Бути готовим відповісти на питання:
- Як забезпечується унікальність лідера в терміні?
- Чому важливо перевіряти актуальність логу кандидата?
- Як працює механізм узгодження логів?
- Які дані потрібно зберігати персистентно?

Тести

1. Що відбувається, коли послідовник не отримує сигнал підтвердження життя від лідера?
- 1) Негайно стає лідером
- 2) Переходить у стан кандидат після часу очікування виборів
- 3) Продовжує чекати необмежено
- 4) Вимикається

2. Скільки голосів потрібно кандидату для перемоги на виборах?
- 1) Один голос
- 2) Всі голоси
- 3) Більшість голосів (включаючи свій)
- 4) Третина голосів

3. Коли лідер може вважати запис закомміченим?
- 1) Одразу після додавання до свого логу

- 2) Коли більшість вузлів підтвердили збереження
- 3) Після відправки клієнту
- 4) Через фіксований час

4. Що перевіряє вузол перед голосуванням за кандидата?
- 1) Тільки термін кандидата
- 2) Термін та актуальність логу кандидата
- 3) Тільки ID кандидата
- 4) Нічого не перевіряє

5. Яка роль heartbeat повідомлень у Raft?
- 1) Тільки для моніторингу
- 2) Запобігання новим виборам та підтвердження лідерства
- 3) Синхронізація часу
- 4) Передача даних

Відповіді: 1-b, 2-c, 3-b, 4-b, 5-b

Тести підвищеної складності

1. Чому Raft використовує випадкові часу очікування виборів?
- 1) Для безпеки
- 2) Для зменшення ймовірності розділеного голосування
- 3) Для економії ресурсів
- 4) Це помилка дизайну

2. Що станеться якщо лідер отримає RequestVote з вищим терміном?
- 1) Проігнорує запит
- 2) Стане послідовником та оновить термін
- 3) Залишиться лідером
- 4) Вимкнеться

3. Як Raft запобігає комміту записів з попередніх термінів?
- 1) Видаляє старі записи
- 2) Лідер може коммітити тільки записи свого терміну
- 3) Використовує додаткове голосування
- 4) Не запобігає

4. Що таке “figure 8” проблема в контексті Raft?

- 1) Проблема з продуктивністю
- 2) Ситуація коли старі записи можуть бути помилково закоммічені
- 3) Мережева топологія
- 4) Тип візуалізації

5. Яка складність пошуку точки розбіжності логів?

- 1) O(1)
- 2) O(log n)
- 3) O(n) в гіршому випадку
- 4) $O(n^2)$

6. Чому pre-vote оптимізація корисна?

- 1) Прискорює вибори
- 2) Запобігає непотрібному збільшенню термінів
- 3) Зменшує кількість повідомлень
- 4) Покращує безпеку

7. Як порожній запис допомагає новому лідеру?

- 1) Тестує мережу
- 2) Дозволяє швидше коммітити записи попередніх термінів
- 3) Зменшує навантаження
- 4) Не має користі

8. Що відбувається при одночасному старті кількох виборів?

- 1) Система блокується
- 2) Можливий розподілений вибір, потрібен новий раунд
- 3) Обирається кілька лідерів
- 4) Вибори скасовуються

9. Як спільний консенсус допомагає при зміні конфігурації?

- 1) Прискорює процес
- 2) Запобігає існуванню двох лідерів під час переходу
- 3) Зменшує кількість повідомлень
- 4) Спрощує реалізацію

10. Яка основна відмінність лініаризованих читань у Raft?

?

- 1) Швидші за звичайні

- 2) Вимагають підтвердження від більшості

- 3) Не потребують диску

- 4) Працюють без лідера

Відповіді:

1. 2) Випадковість зменшує ймовірність того, що кілька вузлів стануть кандидатами одночасно

2. 2) Вищий термін означає новішу інформацію, вузол повинен стати послідовником

3. 2) Лідер коммітить лише записи створені в його терміні, що гарантує безпеку

4. 2) Проблема описана в статті, коли старі записи можуть бути помилково закоммічені

5. 3) У гіршому випадку потрібно перевірити весь лог

6. 2) Вузол перевіряє чи може виграти перед збільшенням терміну

7. 2) Непорожній запис поточного терміну дозволяє безпечно коммітити старі записи

8. 2) Голоси розділяються, жоден кандидат не отримує більшість

9. 2) Використання двох конфігурацій одночасно запобігає розділенню

10. 2) Для гарантії актуальності лідер повинен підтвердити своє лідерство

Завдання для самостійної роботи

1. Реалізуйте Byzantine Fault Tolerant версію Raft з використанням криптографічних підписів.

2. Створіть адаптивний механізм election timeout, що налаштовується на основі статистики мережі.

3. Розробіть ефективний алгоритм компактування логу з мінімальним впливом на продуктивність.

4. Реалізуйте multi-group Raft для шардування даних між кількома Raft групами.

5. Створіть механізм автоматичного відновлення та перебалансування при відмовах вузлів.

6. Дослідіть та реалізуйте різні стратегії пакетування для оптимізації пропускної здатності.

7. Розробіть систему моніторингу та алертів для Raft кластера.

8. Реалізуйте географічно розподілений Raft з оптимізацією для WAN.

9. Створіть формальну верифікацію коректності вашої реалізації Raft.

Теми лабораторних робіт

1. Базова реалізація Raft та тестування сценаріїв відмов.

2. Розробка розподіленого ключа сховища на основі Raft.

3. Порівняльний аналіз продуктивності Raft та Multi-Paxos.

4. Реалізація динамічних змін членства в кластері Raft.

5. Створення Raft-based системи розподілених блокувань.

6. Інтеграція Raft в систему обробки потокових даних.

7. Розробка візуального дебагера для Raft систем.

8. Реалізація Raft з підтримкою читання з послідовників.

9. Створення системи автоматичного тестування Raft реалізацій.

10. Розробка Raft-based координатора для мікросервісної архітектури.

Розподілені хеш-таблиці

Розподілена хеш-таблиця (DHT) — це децентралізована система, що дозволяє ефективно зберігати та знаходити дані у великих, нестабільних мережах, таких як peer-to-peer (P2P) системи. Уявіть її як гігантський телефонний довідник, розподілений між усіма учасниками мережі, де кожен зберігає невелику частину контактів.

На відміну від традиційних систем, де всі дані лежать на одному центральному сервері, у DHT кожен вузол (комп’ютер) мережі відповідає за зберігання певної частини даних. Це досягається за допомогою хеш-функції, яка перетворює ключ (наприклад, назву файлу) у унікальний ідентифікатор (хеш). Цей ідентифікатор потім використовується для визначення, який саме вузол у мережі повинен зберігати відповідні дані (наприклад, сам файл або інформацію про його місцезнаходження).

Основні властивості DHT:

- **Децентралізація:** Відсутній єдиний центральний сервер. Система продовжує працювати, навіть якщо деякі вузли виходять з ладу або покидають мережу.

- **Масштабованість:** DHT ефективно працюють у мережах з тисячами і навіть мільйонами учасників. Додавання нових вузлів не призводить до значного падіння продуктивності.

- **Відмовостійкість:** Системи на основі DHT є стійкими до збоїв. Дані часто дублюються на кількох вузлах, що забезпечує їх доступність навіть при відключенні частини мережі.

- **Ефективний пошук:** Кожен вузол знає лише про невелику кількість інших вузлів, але спеціальні алгоритми маршрутизації дозволяють швидко (зазвичай за логарифмічний час) знайти потрібний вузол, що відповідає за певний ключ.

Найвідомішим прикладом використання технології DHT є BitTorrent. Коли ви завантажуєте файл через торрент, ваш клієнт використовує DHT для пошуку інших користувачів (пірів), у яких є частини потрібного вам файлу, без необхідності звертатися до центрального трекера.

Розподілені хеш-таблиці Chord

Теоретичні відомості

Chord є одним з найвідоміших протоколів для побудови розподілених хеш-таблиць (DHT), розроблений у 2001 році дослідниками з MIT. Цей протокол забезпечує ефективний пошук даних у peer-to-peer мережах, де кожен вузол зберігає лише частину загальної інформації про систему, але при цьому може знайти будь-який ключ за логарифмічний час.

Основні концепції Chord

Chord організовує вузли та ключі в логічне кільце з використанням консистентного хешування. Кожному вузлу та ключу призначається m-бітний ідентифікатор за допомогою криптографічної хеш-функції (наприклад, SHA-1). Простір ідентифікаторів утворює кільце модуло 2^m, де ідентифікатори впорядковані за годинниковою стрілкою від 0 до $2^m - 1$.

```
struct ChordNode {
    nodeId: integer           // m-бітний ідентифікатор вузла
    predecessor: NodeReference // посилання на попередника
    finger: array[m] of NodeReference // finger table
```

```
    successorList: array[r] of NodeReference // список наступників дл
я відмовостійкості
    data: HashMap<Key, Value> // локальні дані
}

struct NodeReference {
    id: integer
    ipAddress: string
    port: integer
}
```

Консистентне хешування

Консистентне хешування є фундаментальною концепцією Chord. Воно забезпечує рівномірний розподіл навантаження та мінімізує переміщення даних при додаванні або видаленні вузлів.

Ключ k призначається вузлу, чий ідентифікатор дорівнює k або є найменшим ідентифікатором, що перевищує k. Цей вузол називається наступником ключа k і позначається як successor(k).

```
function hash(data: string): integer
    // Використовуємо SHA-1 та беремо перші m біт
    hashValue = SHA1(data)
    return hashValue mod 2^m

function getNodeId(nodeAddress: string): integer
    return hash(nodeAddress)

function getKeyId(key: string): integer
    return hash(key)
```

Finger Table

Для ефективного пошуку кожен вузол n підтримує таблицю маршрутизації, яка називається finger table. Ця таблиця містить m записів, де i-й запис містить ідентифікатор першого вузла s, який наступає за n щонайменше на 2^{i-1} позицій на кільці:

$$finger[i] = successor(n + 2^{i-1}) \bmod 2^m$$

де $1 \leq i \leq m$.

```
function initFingerTable(node: ChordNode)
    for i = 1 to m do
        start = (node.nodeId + 2^(i-1)) mod 2^m
        node.finger[i] = findSuccessor(start)
```

Основна операція в Chord - це пошук наступника для заданого ідентифікатора. Алгоритм використовує finger table для швидкого переміщення по кільцю:

```
function findSuccessor(id: integer): NodeReference
    if id ∈ (nodeId, successor.nodeId] then
        return successor
    else
        // Знаходимо найближчий попередній вузол
        n' = closestPrecedingNode(id)
        return n'.findSuccessor(id)  // рекурсивний виклик на віддале
ному вузлі

function closestPrecedingNode(id: integer): NodeReference
    // Проходимо finger table у зворотному порядку
    for i = m downto 1 do
        if finger[i].nodeId ∈ (nodeId, id) then
            return finger[i]
    return self
```

Коли новий вузол n приєднується до мережі Chord, він повинен виконати кілька кроків для інтеграції:

```
function join(n: ChordNode, knownNode: NodeReference)
    if knownNode != null then
        // Ініціалізуємо finger table
        initFingerTable(n, knownNode)
        // Оновлюємо finger tables інших вузлів
        updateOthers(n)
        // Переміщуємо ключі з наступника
        transferKeys(n)
    else
        // n - перший вузол у мережі
        for i = 1 to m do
            n.finger[i] = n
        n.predecessor = n

function initFingerTable(n: ChordNode, knownNode: NodeReference)
    n.finger[1] = knownNode.findSuccessor(n.nodeId + 1)
    n.predecessor = n.finger[1].predecessor
    n.finger[1].predecessor = n

    for i = 1 to m-1 do
        start = (n.nodeId + 2^i) mod 2^m
        if start ∈ [n.nodeId, n.finger[i].nodeId) then
            n.finger[i+1] = n.finger[i]
```

```
        else
            n.finger[i+1] = knownNode.findSuccessor(start)
```

Після приєднання нового вузла необхідно оновити finger tables існуючих вузлів:

```
function updateOthers(n: ChordNode)
    for i = 1 to m do
        // Знаходимо останній вузол p, чий i-й finger може бути n
        p = findPredecessor(n.nodeId - 2^(i-1) + 1)
        p.updateFingerTable(n, i)

function updateFingerTable(s: NodeReference, i: integer)
    if s.nodeId ∈ [nodeId, finger[i].nodeId) then
        finger[i] = s
        // Оновлюємо попередника
        p = predecessor
        p.updateFingerTable(s, i)
```

Для підтримки коректності структури при динамічних змінах, Chord використовує періодичний процес стабілізації:

```
function stabilize()
    x = successor.predecessor
    if x ∈ (nodeId, successor.nodeId) then
        successor = x
    successor.notify(self)

function notify(n': NodeReference)
    if predecessor == null or n'.nodeId ∈ (predecessor.nodeId, nodeId
) then
        predecessor = n'

function fixFingers()
    // Періодично оновлюємо випадковий finger
    i = random(1, m)
    finger[i] = findSuccessor(nodeId + 2^(i-1))
```

Зберігання та пошук даних

Операції з даними в Chord прості та ефективні:

```
function put(key: string, value: any)
    keyId = getKeyId(key)
    responsibleNode = findSuccessor(keyId)
    responsibleNode.store(keyId, value)

function get(key: string): any
    keyId = getKeyId(key)
```

```
    responsibleNode = findSuccessor(keyId)
    return responsibleNode.retrieve(keyId)

function store(keyId: integer, value: any)
    data[keyId] = value

function retrieve(keyId: integer): any
    return data[keyId]
```

Відмовостійкість

Для забезпечення відмовостійкості Chord використовує кілька механізмів:

- Список наступників: Кожен вузол зберігає список з r наступників замість одного

- Реплікація даних: Дані реплікуються на кілька наступних вузлів

- Періодична стабілізація: Виявляє та виправляє неконсистентності

```
function maintainSuccessorList()
    successorList[0] = finger[1]  // безпосередній наступник
    for i = 1 to r-1 do
        successorList[i] = successorList[i-1].successor

function replicateData(key: string, value: any, replicationFactor: in
teger)
    keyId = getKeyId(key)
    replicas = []
    currentNode = findSuccessor(keyId)

    for i = 0 to replicationFactor-1 do
        currentNode.store(keyId, value)
        replicas.append(currentNode)
        currentNode = currentNode.successor

    return replicas
```

Обробка відмов вузлів

При виявленні відмови вузла система автоматично адаптується:

```
function handleNodeFailure(failedNode: NodeReference)
    if failedNode == successor then
        // Використовуємо наступний вузол зі списку наступників
        for i = 0 to r-1 do
            if successorList[i] != failedNode and isAlive(successorLi
st[i]) then
                successor = successorList[i]
                break
```

```
    // Оновлюємо finger table
    for i = 1 to m do
        if finger[i] == failedNode then
            finger[i] = findSuccessor(nodeId + 2^(i-1))
```

Оптимізації продуктивності

Chord включає кілька оптимізацій для покращення продуктивності:

- Кешування: Вузли можуть кешувати популярні пари ключ-значення

- Паралельні запити: Для прискорення пошуку можна відправляти запити кільком вузлам одночасно

- Проксимальна маршрутизація: Враховування мережевої відстані при виборі маршруту

```
function lookupWithCache(key: string): any
    keyId = getKeyId(key)

    // Перевіряємо локальний кеш
    if cache.contains(keyId) then
        return cache.get(keyId)

    // Виконуємо звичайний пошук
    value = get(key)

    // Кешуємо результат
    cache.put(keyId, value, TTL)
    return value
```

Область застосування

Chord та інші розподілені хеш-таблиці знаходять широке застосування в різних сферах:

- Peer-to-peer файлообмінні мережі: BitTorrent DHT використовує модифікований протокол Kademlia для пошуку пірів без центральних трекерів.

- Розподілені системи зберігання: Amazon Dynamo використовує принципи консистентного хешування для розподілу даних між вузлами.

- Content Delivery Networks (CDN): Для ефективного розміщення та пошуку кешованого контенту.

- Розподілені обчислення: Координація задач та розподіл навантаження в grid-системах.

- Децентралізовані соціальні мережі: Зберігання профілів та повідомлень без центральних серверів.

- Internet of Things (IoT): Організація великої кількості пристроїв у самоорганізовані мережі.

Приклади застосування

Приклад 1: Базова робота Chord

Розглянемо Chord-мережу з m=6 (простір ідентифікаторів 0-63) та чотирма вузлами:

Початкова конфігурація:

- Вузол N8 (id=8)

- Вузол N14 (id=14)

- Вузол N21 (id=21)

- Вузол N32 (id=32)

Finger table для N8:

```
i | start | interval    | successor
--|-------|-------------|----------
1 | 9     | [9,10)      | N14
2 | 10    | [10,12)     | N14
3 | 12    | [12,16)     | N14
4 | 16    | [16,24)     | N21
5 | 24    | [24,40)     | N32
6 | 40    | [40,8)      | N8
```

Пошук ключа K30:

1. Клієнт звертається до N8

2. N8 перевіряє: $30 \notin (8,14]$, тому шукає в finger table

3. N8 знаходить finger[5]=N32 як найближчий попередній вузол

4. N8 пересилає запит до N21

5. N21 перевіряє: $30 \in (21,32]$, отже successor(30)=N32

6. N32 відповідає за K30

Приклад 2: Приєднання нового вузла

Новий вузол N26 приєднується до мережі:

Крок 1: Ініціалізація

1. N26 знає про N8

2. N26 запитує у N8: findSuccessor(26)

3. Через finger tables знаходиться N32

4. N26.successor = N32 5. N26.predecessor = N21

Крок 2: Оновлення існуючих вузлів

1. N21.successor змінюється з N32 на N26

2. N32.predecessor змінюється з N21 на N26

Крок 3: Побудова finger table N26

```
finger[1] = successor(27) = N32
finger[2] = successor(28) = N32
finger[3] = successor(30) = N32
finger[4] = successor(34) = N8
finger[5] = successor(42) = N8
finger[6] = successor(58) = N8
```

Крок 4: Переміщення даних Ключі з діапазону (21,26] переміщуються з N32 до N26.

Приклад 3: Обробка відмови вузла

Припустимо, N21 відмовляє:

Виявлення відмови:

1. N14 під час стабілізації не може зв’язатися з N21

2. N14 використовує список наступників і встановлює successor=N26

Відновлення даних:

1. Якщо використовувалася реплікація з фактором 3

2. Дані N21 були також на N26 та N32

3. N26 стає основним власником цих даних

Оновлення finger tables:

1. Вузли, що мали N21 у finger tables, оновлюють записи

2. Через процес fixFingers() всі посилання на N21 замінюються на N26

Переваги та недоліки

Переваги

- Масштабованість: Логарифмічна складність пошуку O(log N) та розмір таблиці маршрутизації O(log N).

- Простота: Відносно простий протокол порівняно з іншими DHT як Pastry або CAN.

- Самоорганізація: Автоматична адаптація до змін у топології мережі.

- Балансування навантаження: Консистентне хешування забезпечує рівномірний розподіл даних.

- Відмовостійкість: Робота продовжується при відмові вузлів завдяки реплікації та альтернативним маршрутам.

- Децентралізація: Відсутність центральних точок відмови або контролю.

Недоліки

- Відсутність врахування мережевої близькості: Chord не враховує фізичну топологію мережі.

- Затримки при стабілізації: Після масових змін потрібен час для стабілізації структури.

- Вразливість до churn: Висока частота приєднання/відключення вузлів може погіршити продуктивність.

- Обмежена виразність запитів: Підтримується лише точний пошук за ключем.

- Накладні витрати на підтримку: Постійний обмін повідомленнями для стабілізації.

- Проблеми безпеки: Вразливість до атак типу Sybil та Eclipse без додаткових механізмів захисту.

Контрольні запитання

1. Що таке консистентне хешування та яку роль воно відіграє в Chord?

2. Поясніть структуру та призначення finger table у протоколі Chord.

3. Як відбувається пошук наступника для заданого ключа та яка його складність?

4. Опишіть процес приєднання нового вузла до Chord-мережі.

5. Які механізми використовує Chord для забезпечення відмовостійкості?

6. Що таке процес стабілізації та навіщо він потрібен?

7. Як Chord обробляє відмову вузлів у мережі?

8. Чому розмір finger table дорівнює m, незалежно від кількості вузлів?

9. Які основні відмінності між Chord та традиційними хеш-таблицями?

10. Як забезпечується рівномірний розподіл навантаження в Chord?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження розподіленої хеш-таблиці Chord

2. Мета

Вивчити принципи побудови розподілених хеш-таблиць, реалізувати протокол Chord, дослідити характеристики масштабованості та відмовостійкості в динамічному середовищі.

3. Завдання

Реалізувати повнофункціональну Chord DHT з підтримкою основних операцій (lookup, join, leave), механізмами стабілізації та відмовостійкості. Створити візуалізацію топології мережі. Провести експерименти з різними розмірами мережі та частотою churn.

4. Опис

Лабораторна робота передбачає поетапну реалізацію всіх компонентів Chord: консистентного хешування, finger tables, алгоритмів пошуку та стабілізації. Студенти повинні реалізувати як локальну симуляцію, так і розподілену версію з реальною мережевою взаємодією.

5. Порядок виконання

- Реалізувати консистентне хешування та базові структури даних
- Створити алгоритм побудови та оновлення finger table
- Реалізувати функції findSuccessor та closestPrecedingNode
- Додати механізм приєднання нових вузлів
- Реалізувати процеси стабілізації та fixFingers
- Створити операції put/get для зберігання даних
- Додати реплікацію та обробку відмов
- Реалізувати візуалізацію Chord-кільця
- Створити тестове середовище з емуляцією churn
- Провести вимірювання продуктивності

6. Результат

Працююча реалізація Chord DHT з графічним інтерфейсом, статистика про кількість хопів для пошуку, демонстрація роботи при додаванні/видаленні

вузлів, аналіз впливу churn на продуктивність, порівняння з централізованими рішеннями.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Модуль консистентного хешування
- Реалізація finger table та її оновлення
- Алгоритми пошуку та маршрутизації
- Механізми стабілізації та обробки відмов
- Інтерфейс для операцій put/get

2. Продемонструвати роботу додатку:

- Побудова Chord-мережі з N вузлів
- Виконання операцій put/get
- Приєднання нових вузлів з перерозподілом даних
- Коректна робота при відмові вузлів
- Візуалізація топології та маршрутів пошуку

3. Бути готовим відповісти на питання:

- Як працює консистентне хешування?
- Чому finger table має логарифмічний розмір?
- Як забезпечується коректність при паралельних змінах?
- Які компроміси між частотою стабілізації та накладними витратами?

Тести

1. Яка складність пошуку в Chord-мережі з N вузлів?

- 1) O(1)
- 2) O(log N)
- 3) O(N)
- 4) $O(N^2)$

2. Скільки записів містить finger table при m-бітних ідентифікаторах?

- 1) N записів
- 2) 2^m записів
- 3) m записів

- 4) log N записів

3. Який вузол відповідає за ключ k?

- 1) Вузол з найменшим id

- 2) Вузол з id = k

- 3) Перший вузол з id ⩾ k

- 4) Випадковий вузол

4. Що таке процес стабілізації в Chord?

- 1) Видалення старих даних

- 2) Періодичне оновлення посилань між вузлами

- 3) Балансування навантаження

- 4) Шифрування даних

5. Як Chord забезпечує відмовостійкість?

- 1) Центральний сервер резервного копіювання

- 2) Реплікація даних на наступників

- 3) Постійне дублювання всіх запитів

- 4) Використання RAID

Відповіді:

1-2,

2-3,

3-3,

4-2,

5-2

Тести підвищеної складності

1. Чому в Chord використовується SHA-1 для генерації ідентифікаторів?

- 1) Для шифрування даних

- 2) Для рівномірного розподілу в просторі ідентифікаторів

- 3) Для стиснення даних

- 4) Для аутентифікації

2. Яка ймовірність того, що при N вузлах два отримають однаковий id при m-бітних ідентифікаторах?

- 1) 0 при m > log N
- 2) 1 - $e^{(-N^2/2}$(m+1))
- 3) N/2^m
- 4) 1/N

3. Як впливає churn rate на продуктивність Chord?
- 1) Не впливає
- 2) Лінійно погіршує
- 3) Може призвести до тимчасової неконсистентності
- 4) Покращує продуктивність

4. Що таке “ring inconsistency” в контексті Chord?
- 1) Різні вузли мають різні дані
- 2) Порушення інваріанту successor/predecessor
- 3) Повільний пошук
- 4) Втрата даних

5. Яка оптимальна частота виклику stabilize()?
- 1) Кожну секунду
- 2) Залежить від churn rate
- 3) Раз на годину
- 4) При кожному запиті

6. Як virtual nodes покращують балансування навантаження?
- 1) Створюють резервні копії
- 2) Один фізичний вузол відповідає за кілька позицій на кільці
- 3) Прискорюють пошук
- 4) Зменшують розмір finger table

7. Яка основна вразливість Chord до Sybil-атак?
- 1) Повільний пошук
- 2) Зловмисник може контролювати значну частину простору ідентифікаторів
- 3) Втрата даних
- 4) Немає вразливостей

8. Як проксимальна маршрутизація покращує Chord?

- 1) Зменшує кількість хопів

- 2) Враховує мережеву відстань при виборі маршруту

- 3) Збільшує безпеку

- 4) Спрощує реалізацію

9. Що відбувається при одночасному приєднанні двох вузлів між одними сусідами?

- 1) Система падає

- 2) Один стає successor іншого через порівняння id

- 3) Обидва відхиляються

- 4) Створюється новий простір

10. Як Byzantine failures впливають на Chord?

- 1) Не впливають

- 2) Можуть порушити маршрутизацію без додаткових механізмів

- 3) Автоматично виявляються

- 4) Покращують продуктивність

Відповіді:

1. b) SHA-1 забезпечує рівномірний розподіл, що критично для балансування навантаження

2. b) Це формула парадоксу днів народження для хеш-колізій

3. c) Високий churn може призвести до застарілих finger tables та тимчасових помилок маршрутизації

4. b) Коли successor/predecessor посилання не узгоджені, кільце “розривається”

5. b) Оптимальна частота залежить від динаміки мережі - частіше при високому churn

6. b) Virtual nodes розподіляють навантаження одного вузла по кільцю

7. b) Створюючи багато вузлів, зловмисник може контролювати сегменти кільця

8. b) Вибір фізично ближчих вузлів зменшує мережеві затримки

9. b) Детерміністичне впорядкування за id вирішує конфлікт

10. b) Зловмисні вузли можуть навмисно давати неправильні відповіді

Завдання для самостійної роботи

1. Реалізуйте модифікацію Chord з підтримкою range queries використовуючи додаткові структури даних.

2. Створіть гібридну DHT, що поєднує переваги Chord та Kademlia.

3. Розробіть механізм репутації вузлів для захисту від зловмисних учасників.

4. Реалізуйте Chord з підтримкою географічної локальності для оптимізації затримок.

5. Дослідіть вплив різних хеш-функцій на рівномірність розподілу даних.

6. Створіть адаптивний механізм реплікації, що враховує популярність даних.

7. Реалізуйте підтримку Byzantine fault tolerance в Chord.

8. Розробіть систему моніторингу та візуалізації для великих Chord-мереж.

9. Створіть механізм міграції даних для планового виведення вузлів.

10. Дослідіть можливості використання машинного навчання для оптимізації маршрутизації.

Теми лабораторних робіт

1. Базова реалізація Chord DHT з візуалізацією кільця.

2. Порівняльний аналіз різних DHT протоколів (Chord, Kademlia, Pastry).

3. Розробка розподіленої файлової системи на основі Chord.

4. Створення децентралізованої соціальної мережі з використанням Chord.

5. Реалізація відмовостійкого key-value сховища на базі Chord.

6. Інтеграція Chord в торрент-клієнт для distributed tracker.

7. Розробка системи розподілених обчислень з Chord для координації.

8. Створення Chord-based CDN для розподілу контенту.

9. Реалізація приватної DHT з шифруванням та аутентифікацією.

10. Дослідження масштабованості Chord в хмарному середовищі.

Розподілені хеш-таблиці Kademlia

Теоретичні відомості

Kademlia є одним з найефективніших протоколів для розподілених хеш-таблиць (DHT), розроблений Петаром Маймунковим та Девідом Мазьєром у 2002 році. Протокол використовує унікальну метрику відстані на основі операції XOR, що забезпечує симетричність та ефективність маршрутизації. Kademlia став основою для багатьох практичних систем, включаючи BitTorrent DHT, Ethereum та IPFS.

Основні концепції

Кожен вузол та ключ у Kademlia ідентифікується унікальним b-бітним ідентифікатором (зазвичай b=160 для SHA-1). Ключова інновація Kademlia полягає у використанні XOR-метрики для визначення відстані між ідентифікаторами:

$$d(x, y) = x \oplus y$$

де $\oplus$ позначає побітову операцію XOR.

Ця метрика має важливі властивості: - Симетричність: $d(x, y) = d(y, x)$ - Тотожність: $d(x, x) = 0$ - Нерівність трикутника: $d(x, z) \leq d(x, y) + d(y, z)$ - Унарність: для кожного x та відстані d існує рівно один y такий, що $d(x, y) = d$

```
struct KademliaNode {
    nodeId: BitString[b]      // b-бітний ідентифікатор вузла
    kBuckets: array[b] of KBucket  // k-buckets для зберігання контак
тів
    storage: HashMap<Key, Value>    // локальне сховище даних

    // Параметри протоколу
    k: integer = 20              // розмір k-bucket
    alpha: integer = 3           // паралельність пошуку
    b: integer = 160             // довжина ідентифікатора в бітах
}

struct Contact {
    nodeId: BitString[b]
    ipAddress: string
    udpPort: integer
    lastSeen: timestamp
}

struct KBucket {
```

```
    contacts: list of Contact    // максимум k контактів
    replacementCache: list of Contact  // кеш для заміни
}
```

K-Buckets структура

Кожен вузол організовує відомі йому контакти в b різних k-buckets. K-bucket з індексом i містить контакти вузлів, відстань до яких від поточного вузла знаходиться в діапазоні $[2^i, 2^{i+1})$. Таким чином, k-bucket 0 містить вузли на відстані 1, k-bucket 1 - на відстані 2-3, і так далі.

```
function getBucketIndex(localId: BitString, remoteId: BitString): int
eger
    distance = localId XOR remoteId
    if distance == 0 then
        return -1  // той самий вузол

    // Знаходимо позицію старшого встановленого біта
    for i = b-1 downto 0 do
        if distance.bit(i) == 1 then
            return i

    return 0

function addContact(node: KademliaNode, contact: Contact)
    if contact.nodeId == node.nodeId then
        return  // не додаємо себе

    bucketIndex = getBucketIndex(node.nodeId, contact.nodeId)
    bucket = node.kBuckets[bucketIndex]

    // Перевіряємо чи контакт вже існує
    existingIndex = bucket.contacts.indexOf(contact)
    if existingIndex >= 0 then
        // Переміщуємо в кінець (найсвіжіший)
        bucket.contacts.moveToEnd(existingIndex)
        contact.lastSeen = now()
        return

    // Якщо є місце, просто додаємо
    if bucket.contacts.size() < k then
        bucket.contacts.append(contact)
        return

    // Bucket повний, перевіряємо найстаріший контакт
    oldestContact = bucket.contacts[0]
```

```
    if isAlive(oldestContact) then
        // Старий контакт живий, новий йде в кеш
        bucket.replacementCache.append(contact)
        if bucket.replacementCache.size() > k then
            bucket.replacementCache.removeFirst()
    else
        // Заміняємо мертвий контакт
        bucket.contacts.removeFirst()
        bucket.contacts.append(contact)
```

Основні RPC операції

Kademlia використовує чотири основні RPC операції:

PING

Перевіряє чи вузол активний:

```
function handlePing(sender: Contact): PongResponse
    addContact(self, sender)
    return PongResponse(nodeId = self.nodeId)
```

STORE

Зберігає пару ключ-значення на вузлі:

```
function handleStore(sender: Contact, key: BitString, value: any): St
oreResponse
    addContact(self, sender)
    self.storage[key] = value
    return StoreResponse(success = true)
```

FIND_NODE

Повертає k найближчих до цільового ID контактів:

```
function handleFindNode(sender: Contact, targetId: BitString): FindNo
deResponse
    addContact(self, sender)

    closestContacts = []

    // Збираємо контакти з усіх k-buckets
    for bucket in self.kBuckets do
        for contact in bucket.contacts do
            closestContacts.append(contact)

    // Сортуємо за відстанню до targetId
    closestContacts.sortBy(contact => contact.nodeId XOR targetId)

    // Повертаємо k найближчих
    return FindNodeResponse(contacts = closestContacts[0:k])
```

FIND_VALUE

Схожий на FIND_NODE, але повертає значення якщо воно є:

```
function handleFindValue(sender: Contact, key: BitString): FindValueR
esponse
    addContact(self, sender)

    // Перевіряємо чи маємо значення
    if key in self.storage then
        return FindValueResponse(value = self.storage[key])
    else
        // Повертаємо найближчі контакти як у FIND_NODE
        return FindValueResponse(contacts = handleFindNode(sender, ke
y).contacts)
```

Алгоритм пошуку вузла

Пошук у Kademlia виконується ітеративно, запитуючи все ближчі вузли до цілі:

```
function nodeLookup(targetId: BitString): list of Contact
    // Ініціалізація
    shortlist = getClosestContacts(targetId, alpha)
    closestNode = shortlist[0]
    closestDistance = closestNode.nodeId XOR targetId
    contacted = set()
    active = set()

    while true do
        // Вибираємо alpha непроконтактованих вузлів
        toContact = []
        for contact in shortlist do
            if contact not in contacted and contact not in active the
n
                toContact.append(contact)
                if toContact.size() >= alpha then
                    break

        if toContact.isEmpty() then
            break  // більше немає кого контактувати

        // Паралельно відправляємо запити
        for contact in toContact do
            active.add(contact)
            sendAsync(contact, FIND_NODE(targetId), callback = (respo
nse) => {
                active.remove(contact)
```

```
                contacted.add(contact)

                if response != null then
                    for newContact in response.contacts do
                        distance = newContact.nodeId XOR targetId
                        if distance < closestDistance then
                            closestDistance = distance
                            closestNode = newContact

                        // Додаємо до shortlist якщо ближче за k-й ел
емент
                        if shortlist.size() < k or distance < (shortl
ist[k-1].nodeId XOR targetId) then
                            shortlist.append(newContact)
                            shortlist.sortBy(c => c.nodeId XOR target
Id)
                            shortlist = shortlist[0:k]
            })

        // Чекаємо на відповіді
        waitForActiveRequests()

        // Перевіряємо умову завершення
        if not foundCloserInLastRound then
            // Контактуємо всі k найближчих вузлів
            for contact in shortlist[0:k] do
                if contact not in contacted then
                    sendSync(contact, FIND_NODE(targetId))
            break

    return shortlist[0:k]
```

Зберігання та отримання даних

Операції put та get використовують алгоритм пошуку вузла:

```
function put(key: string, value: any)
    keyId = hash(key)  // SHA-1 або інша хеш-функція

    // Знаходимо k найближчих вузлів до ключа
    closestNodes = nodeLookup(keyId)

    // Зберігаємо на всіх k вузлах
    for node in closestNodes do
        sendAsync(node, STORE(keyId, value))

function get(key: string): any
```

```
    keyId = hash(key)

    // Ініціалізація як у nodeLookup
    shortlist = getClosestContacts(keyId, alpha)
    contacted = set()
    foundValue = null

    while foundValue == null and hasMoreToContact() do
        toContact = selectAlphaClosestUncontacted(shortlist)

        for contact in toContact do
            response = sendSync(contact, FIND_VALUE(keyId))
            contacted.add(contact)

            if response.hasValue then
                foundValue = response.value
                // Кешуємо на вузлах, що були ближчими але не мали зн
ачення
                cacheOnCloserNodes(keyId, foundValue, shortlist, cont
act)
                break
            else
                updateShortlist(shortlist, response.contacts)

    return foundValue
```

Приєднання до мережі

Процес приєднання нового вузла до Kademlia мережі:

```
function join(bootstrapNode: Contact)
    // Додаємо bootstrap вузол
    addContact(self, bootstrapNode)

    // Виконуємо пошук самого себе
    nodeLookup(self.nodeId)

    // Оновлюємо всі k-buckets
    refreshBuckets()

function refreshBuckets()
    // Для кожного k-bucket виконуємо пошук випадкового ID в його діа
пазоні
    for i = 0 to b-1 do
        if self.kBuckets[i].isEmpty() then
            // Генеруємо випадковий ID на відстані 2^i від нас
            randomId = self.nodeId XOR (1 << i)
```

```
            // Додаємо випадкові біти для молодших розрядів
            for j = 0 to i-1 do
                if random() < 0.5 then
                    randomId = randomId XOR (1 << j)

            nodeLookup(randomId)
```

Обслуговування k-buckets

Kademlia використовує кілька механізмів для підтримки актуальності k-buckets:

```
function maintainBuckets()
    // Періодично для кожного bucket
    for bucket in self.kBuckets do
        // Перевіряємо старі контакти
        for contact in bucket.contacts do
            if now() - contact.lastSeen > STALE_TIMEOUT then
                pingResponse = sendSync(contact, PING())
                if pingResponse == null then
                    // Контакт мертвий, заміняємо з кешу
                    bucket.contacts.remove(contact)
                    if not bucket.replacementCache.isEmpty() then
                        newContact = bucket.replacementCache.popFirst
()
                        bucket.contacts.append(newContact)

function handleIncomingMessage(sender: Contact, message: any)
    // Оновлюємо контакт при будь-якому вхідному повідомленні
    addContact(self, sender)

    // Обробляємо повідомлення
    switch message.type do
        case PING: return handlePing(sender)
        case STORE: return handleStore(sender, message.key, message.v
alue)
        case FIND_NODE: return handleFindNode(sender, message.targetI
d)
        case FIND_VALUE: return handleFindValue(sender, message.key)
```

Реплікація та відновлення даних

Kademlia автоматично забезпечує реплікацію даних:

```
function republishData()
    // Кожну годину переопубліковуємо дані
    for (key, value) in self.storage do
        if now() - value.publishTime > REPUBLISH_INTERVAL then
            put(key, value.data)
```

```
            value.publishTime = now()

function replicateData()
    // При появі нового близького вузла
    for (key, value) in self.storage do
        keyId = BitString(key)
        closestNodes = getKClosestNodes(keyId)

        for node in closestNodes do
            if isNewOrCloser(node, keyId) then
                sendAsync(node, STORE(keyId, value))
```

Оптимізації протоколу

Kademlia включає кілька оптимізацій для покращення продуктивності:

1. Паралельні запити

Використання параметра alpha для паралельних запитів прискорює пошук:

```
function optimizedLookup(targetId: BitString): list of Contact
    pipeline = []
    results = []

    // Запускаємо alpha паралельних потоків
    for i = 0 to alpha-1 do
        pipeline.append(LookupThread(targetId, i))

    // Збираємо результати
    while not allThreadsComplete(pipeline) do
        for thread in pipeline do
            if thread.hasNewResults() then
                results.merge(thread.getResults())
                updateAllThreads(pipeline, results)

    return results.getKClosest(k)
```

2. Кешування популярних значень

Вузли кешують популярні значення на шляху запитів:

```
function cacheOnPath(key: BitString, value: any, pathNodes: list of C
ontact)
    // Визначаємо експоненційний час кешування
    for i = 0 to pathNodes.size()-1 do
        node = pathNodes[i]
        // Ближчі вузли кешують довше
        cacheDuration = BASE_CACHE_TIME * 2^(pathNodes.size() - i)
        sendAsync(node, STORE_CACHE(key, value, cacheDuration))
```

Область застосування

Kademlia знайшов широке застосування в різних децентралізованих системах:

- Файлообмінні мережі: BitTorrent DHT обслуговує мільйони користувачів для пошуку пірів без центральних трекерів.

- Блокчейн системи: Ethereum використовує модифікований Kademlia для пошуку вузлів та поширення транзакцій.

- Децентралізовані сховища: IPFS (InterPlanetary File System) базується на Kademlia для адресації контенту.

- Комунікаційні системи: Tox messenger використовує Kademlia для встановлення з'єднань між користувачами.

- CDN та розподілені кеші: Для ефективного розміщення та пошуку кешованого контенту.

- IoT мережі: Самоорганізація великої кількості пристроїв без центрального сервера.

Приклади застосування

Приклад 1: Базова робота Kademlia

Розглянемо мережу з 4-бітними ідентифікаторами (простір 0-15) та k=2:

Вузли в мережі: - Вузол A: ID = 0010 (2) - Вузол B: ID = 0101 (5) - Вузол C: ID = 1000 (8) - Вузол D: ID = 1110 (14)

K-buckets для вузла A (ID=2):

```
Bucket 0 (відстань 1): [B(5)] // 2 XOR 5 = 7 = 0111
Bucket 1 (відстань 2-3): []
Bucket 2 (відстань 4-7): [B(5)]
Bucket 3 (відстань 8-15): [C(8), D(14)]
```

Пошук вузла з ID=12 від вузла A: 1. A обчислює XOR відстані до відомих вузлів: - До B: 12 XOR 5 = 9 - До C: 12 XOR 8 = 4 - До D: 12 XOR 14 = 2 2. A контактує D (найближчий) 3. D повертає свої контакти, включаючи можливо ближчі вузли 4. Процес продовжується до знаходження k найближчих

Приклад 2: Додавання нового вузла

Новий вузол E з ID=11 приєднується через вузол A:

Процес приєднання: 1. E додає A до своїх k-buckets 2. E виконує nodeLookup(11) через A 3. A повертає найближчі відомі вузли: C(8) та D(14) 4. E контактує C та D 5. E отримує їхні контакти та будує свої k-buckets:

```
Bucket 0: [C(8)]  // 11 XOR 8 = 3    Bucket 1: [D(14)] // 11 XOR 14 = 5    Bucket 3: [A(2), B(5)]
```

Приклад 3: Зберігання та пошук даних

Зберігання ключа “hello” зі значенням “world”:

Процес зберігання: 1. Обчислюємо keyId = hash(“hello”) = 13 (припустимо) 2. Виконуємо nodeLookup(13) для знаходження k найближчих вузлів 3. Найближчі вузли до 13: - D(14): відстань = 13 XOR 14 = 3 - E(11): відстань = 13 XOR 11 = 6 4. Зберігаємо на D та E

Процес пошуку: 1. Інший вузол шукає “hello” 2. Обчислює keyId = 13 3. Виконує iterativeFindValue(13) 4. Знаходить значення на D або E

Приклад 4: Обробка відмов та відновлення

Сценарій відмови вузла C(8):

Виявлення відмови: 1. Вузли періодично пінгують контакти зі своїх k-buckets 2. A виявляє, що C не відповідає 3. A видаляє C з k-bucket та додає контакт з replacementCache

Відновлення даних: 1. Дані, що зберігалися на C, мають копії на k-1 інших вузлах 2. При виявленні зникнення C, сусідні вузли перереплікують дані 3. Наприклад, якщо C зберігав ключ з ID=9, найближчі живі вузли (D, E) забезпечать k копій

Приклад 5: Масштабування мережі

Розглянемо зростання Kademlia мережі:

10 вузлів (b=160, k=20): - Середня кількість хопів: ~3 - Розмір routing table: ~30 контактів - Ймовірність втрати даних при одночасній відмові 10 вузлів: ~0%

1 мільйон вузлів: - Середня кількість хопів: $\log_2(1M) \approx 20$ - Розмір routing table: ~400 контактів - Навантаження на вузол: 0.0001% всіх даних

Переваги масштабування: - Логарифмічне зростання складності - Константний розмір локального стану - Автоматичне балансування навантаження

Переваги та недоліки

Переваги

- **XOR-метрика**: Забезпечує унікальні властивості симетричності та унарності, що спрощує аналіз та реалізацію.

- Ефективність пошуку: O(log n) повідомлень для пошуку серед n вузлів з високою ймовірністю.

- Самоорганізація: Автоматична адаптація до змін топології без центральної координації.

- Стійкість до churn: K-buckets природно надають перевагу стабільним вузлам.

- Паралелізм: Можливість паралельних запитів прискорює операції.

- Простота реалізації: Менше станів та простіші інваріанти порівняно з іншими DHT.

Недоліки

- Відсутність безпеки: Базовий протокол вразливий до Sybil та Eclipse атак.

- Накладні витрати на підтримку: Постійний обмін повідомленнями для підтримки k-buckets.

- Обмежені запити: Підтримує лише точний пошук за ключем, не підтримує range queries.

- Залежність від якості хешування: Нерівномірне хешування призводить до дисбалансу.

- Проблеми з NAT: Утруднена робота за NAT без додаткових механізмів.

- Відсутність стимулів: Немає вбудованих механізмів заохочення чесної поведінки.

Контрольні запитання

1. Які властивості має XOR-метрика в Kademlia та чому вони важливі?

2. Як організовані k-buckets та який принцип їх заповнення?

3. Опишіть алгоритм ітеративного пошуку вузла в Kademlia.

4. Яка різниця між операціями FIND_NODE та FIND_VALUE?

5. Як Kademlia забезпечує стійкість до відмов вузлів?

6. Що таке параметр alpha та як він впливає на продуктивність?

7. Як відбувається процес приєднання нового вузла до мережі?

8. Чому Kademlia надає перевагу старим контактам у k-buckets?

9. Як працює механізм кешування популярних значень?

10. Які основні параметри протоколу (k, alpha, b) та їх типові значення?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження розподіленої хеш-таблиці Kademlia

2. Мета

Вивчити принципи роботи DHT на основі XOR-метрики, реалізувати протокол Kademlia, дослідити характеристики продуктивності та стійкості до відмов у динамічному середовищі.

3. Завдання

Реалізувати повнофункціональну Kademlia DHT з усіма базовими операціями, механізмами обслуговування k-buckets та реплікацією даних. Створити візуалізацію XOR-простору та маршрутів пошуку. Провести порівняння з іншими DHT протоколами.

4. Опис

Лабораторна робота передбачає поетапну реалізацію компонентів Kademlia: XOR-метрики, k-buckets структури, RPC операцій, алгоритмів пошуку. Студенти реалізують як симуляцію для вивчення властивостей, так і мережеву версію для практичного застосування.

5. Порядок виконання

- Реалізувати XOR-метрику та операції з бітовими рядками
- Створити структуру k-buckets з правильною логікою оновлення
- Реалізувати чотири базові RPC операції
- Розробити алгоритм ітеративного пошуку вузлів
- Додати операції зберігання та отримання даних
- Реалізувати механізм приєднання нових вузлів
- Створити процедури обслуговування та реплікації
- Розробити візуалізацію XOR-простору
- Реалізувати симулятор churn
- Провести експерименти та зібрати статистику

6. Результат

Працююча реалізація Kademlia DHT з GUI для моніторингу, статистика розподілу даних та навантаження, демонстрація роботи при різних

сценаріях відмов, аналіз впливу параметрів k та alpha на продуктивність, порівняння з Chord DHT.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Модуль XOR-метрики та операцій з ID
- Реалізація k-buckets з replacement cache
- RPC операції PING, STORE, FIND_NODE, FIND_VALUE
- Алгоритм ітеративного пошуку
- Механізми реплікації та обслуговування

2. Продемонструвати роботу додатку:

- Побудова Kademlia мережі з візуалізацією
- Зберігання та пошук даних з трасуванням маршруту
- Приєднання/відключення вузлів в реальному часі
- Робота при масових відмовах вузлів
- Статистика хопів та затримок

3. Бути готовим відповісти на питання:

- Чому XOR-метрика краща за звичайну числову відстань?
- Як вибір k впливає на надійність та продуктивність?
- Чому старі вузли мають перевагу в k-buckets?
- Як Kademlia протидіє eclipse-атакам?

Тести

1. Яка складність пошуку в Kademlia мережі з n вузлів?

- 1) O(1)
- 2) O(log n)
- 3) O(√n)
- 4) O(n)

2. Скільки контактів зберігає кожен k-bucket?

- 1) Необмежено
- 2) Рівно k
- 3) Максимум k

- 4) Мінімум k

3. Яка властивість XOR-метрики є унікальною?

- 1) Симетричність
- 2) Для кожної відстані існує лише одна точка
- 3) Нерівність трикутника
- 4) Транзитивність

4. Що повертає FIND_VALUE якщо значення не знайдено?

- 1) Помилку
- 2) Null
- 3) K найближчих вузлів
- 4) Порожній список

5. Коли новий контакт замінює старий у k-bucket?

- 1) Завжди
- 2) Ніколи
- 3) Коли старий не відповідає на ping
- 4) Випадково

Відповіді:

1-2,

2-3,

3-2,

4-3,

5-3

Тести підвищеної складності

1. Чому Kademlia використовує префіксні таблиці маршрутизації?

- 1) Для економії пам’яті
- 2) Це природно випливає з XOR-метрики
- 3) Для швидшого пошуку
- 4) Для безпеки

2. Яка ймовірність що конкретний k-bucket буде порожній?

- 1) 0

- 2) 1/2
- 3) $(1-1/2^{i)}n$ для bucket i при n вузлах
- 4) 1/n

3. Як alpha впливає на надійність пошуку?
- 1) Не впливає
- 2) Збільшує ймовірність успіху при втратах пакетів
- 3) Зменшує надійність
- 4) Впливає тільки на швидкість

4. Що таке “iterative routing” в контексті Kademlia?
- 1) Рекурсивні виклики між вузлами
- 2) Ініціатор контролює весь процес пошуку
- 3) Автоматична маршрутизація
- 4) Випадковий пошук

5. Чому інтервал повторної публікації зазвичай 1 година?
- 1) Стандарт протоколу
- 2) Баланс між накладними витратами та надійністю
- 3) Обмеження мережі
- 4) Історичні причини

6. Як Kademlia протидіє “розрідженню” ID-простору?
- 1) Використовує віртуальні вузли
- 2) Refresh buckets для порожніх діапазонів
- 3) Динамічно змінює b
- 4) Не протидіє

7. Яка складність пам’яті для зберігання стану маршрутизації?
- 1) O(n)
- 2) O(log n)
- 3) O(k × log n)
- 4) O(1)

8. Що відбувається при одночасному STORE від різних вузлів?
- 1) Конфлікт, система падає

- 2) Зберігається останнє значення
- 3) Зберігаються обидва з різними версіями
- 4) Перше значення блокує інші

9. Як sibling list покращує надійність?
- 1) Дублює всі дані
- 2) Зберігає k найближчих вузлів для швидкого відновлення
- 3) Створює ієрархію вузлів
- 4) Шифрує з’єднання

10. Чому BitTorrent використовує модифіковану версію Kademlia?
- 1) Для підтримки магнет-посилань
- 2) Оригінальна версія повільна
- 3) Для інтеграції з трекерами
- 4) Патентні обмеження

Відповіді:

1. 2) XOR-метрика природно створює префіксну структуру через біти
2. 3) Ймовірність експоненційно зменшується для дальніх buckets
3. 2) Паралельні запити збільшують шанс отримати відповідь
4. 2) На відміну від рекурсивного, ініціатор сам керує процесом
5. 2) Компроміс між підтримкою доступності та трафіком
6. 2) Активний пошук у порожніх діапазонах заповнює k-buckets
7. 3) k контактів для кожного з log n buckets
8. 2) Немає консенсусу, останнє значення перезаписує
9. 2) Список k найближчих дозволяє швидко знайти заміну
10. 1) DHT використовується для пошуку пірів за info-hash

Завдання для самостійної роботи

1. Реалізуйте S/Kademlia з додатковими механізмами безпеки проти атак.
2. Створіть гібридну DHT, що комбінує переваги Kademlia та Chord.
3. Розробіть механізм підтримки range queries в Kademlia.
4. Реалізуйте географічно-оптимізовану версію Kademlia.
5. Дослідіть вплив різних хеш-функцій на розподіл навантаження.

6. Створіть адаптивний механізм вибору параметрів k та alpha.

7. Реалізуйте стійку до Byzantine failures версію Kademlia.

8. Розробіть інструменти візуалізації та аналізу великих Kademlia мереж.

9. Створіть оптимізовану для мобільних пристроїв версію протоколу.

10. Дослідіть можливості використання машинного навчання для оптимізації маршрутизації.

Теми лабораторних робіт

1. Базова реалізація Kademlia DHT з візуалізацією XOR-простору.

2. Порівняльний аналіз Kademlia, Chord та Pastry DHT.

3. Розробка децентралізованої файлової системи на основі Kademlia.

4. Створення P2P месенджера з використанням Kademlia для NAT traversal.

5. Реалізація приватної Kademlia мережі з шифруванням.

6. Інтеграція Kademlia в blockchain для network discovery.

7. Розробка CDN з використанням Kademlia для маршрутизації.

8. Створення розподіленої бази даних на основі Kademlia.

9. Реалізація стійкої до цензури системи публікації контенту.

10. Дослідження масштабованості Kademlia в умовах глобальної мережі.

13.4. Парадигми обчислень: MapReduce.

Теоретичні відомості

MapReduce є програмною моделлю та парадигмою обчислень для обробки великих обсягів даних у розподілених системах. Розроблена в Google у 2004 році Джеффрі Діном та Санджаєм Гемаватом, ця модель дозволяє розробникам писати прості програми, які автоматично паралелізуються та виконуються на кластерах з тисяч машин. Основна ідея MapReduce полягає в розділенні обчислювальної задачі на дві основні фази: Map (відображення) та Reduce (згортка), між якими відбувається фаза Shuffle (перемішування).

MapReduce базується на функціональному програмуванні, де обчислення представляються як композиція функцій вищого порядку. Модель працює з парами ключ-значення та складається з трьох основних етапів:

- **Map фаза**: Застосовує функцію map до кожного вхідного запису, генеруючи проміжні пари ключ-значення

- **Shuffle фаза**: Групує всі проміжні значення за ключами

- **Reduce фаза**: Агрегує значення для кожного унікального ключа

```
// Абстрактна модель MapReduce
Input: список пар (k1, v1)
Map: (k1, v1) → список пар (k2, v2)
Shuffle: (k2, список[v2]) для кожного унікального k2
Reduce: (k2, список[v2]) → список пар (k3, v3)
Output: список пар (k3, v3)
```

Система MapReduce складається з кількох ключових компонентів:

```
struct MapReduceJob {
    inputPath: string              // шлях до вхідних даних
    outputPath: string             // шлях для результатів
    mapFunction: function          // користувацька map функція
    reduceFunction: function       // користувацька reduce функція
    combinerFunction: function     // опціональна функція для локальн
ої агрегації
    partitionerFunction: function  // функція розподілу ключів між re
ducer'ами
    numMappers: integer           // кількість map задач
    numReducers: integer          // кількість reduce задач
}

struct MapTask {
    taskId: integer
    inputSplit: DataSplit         // частина вхідних даних
    mapFunction: function
    outputPartitions: array[Partition]  // розділені проміжні дані
}

struct ReduceTask {
    taskId: integer
    partitionId: integer
    reduceFunction: function
    inputData: array[ShuffledData]     // дані від всіх mapper'ів
}
```

Map фаза

Map фаза обробляє вхідні дані паралельно на багатьох вузлах:

```
function runMapPhase(job: MapReduceJob, inputSplits: array[DataSplit]
)
    mapTasks = []
```

```
    // Створюємо map задачі для кожного input split
    for i = 0 to inputSplits.length - 1 do
        task = MapTask(
            taskId = i,
            inputSplit = inputSplits[i],
            mapFunction = job.mapFunction
        )
        mapTasks.append(task)

    // Виконуємо map задачі паралельно
    results = parallelExecute(mapTasks, executeMapTask)
    return results

function executeMapTask(task: MapTask): array[Partition]
    // Ініціалізуємо партиції для кожного reducer
    partitions = array[numReducers]
    for i = 0 to numReducers - 1 do
        partitions[i] = new Partition()

    // Обробляємо кожен запис у input split
    for record in task.inputSplit do
        key1 = record.key
        value1 = record.value

        // Викликаємо користувацьку map функцію
        intermediateResults = task.mapFunction(key1, value1)

        // Розподіляємо результати по партиціях
        for (key2, value2) in intermediateResults do
            partitionId = partitioner(key2, numReducers)
            partitions[partitionId].add(key2, value2)

    // Сортуємо кожну партицію за ключем
    for partition in partitions do
        partition.sortByKey()

    // Зберігаємо партиції на диск
    for i = 0 to partitions.length - 1 do
        savePartitionToDisk(task.taskId, i, partitions[i])

    return partitions
```

Функція Partitioner

Partitioner визначає, який reducer отримає конкретний ключ:

```
function defaultPartitioner(key: any, numReducers: integer): integer
    // Використовуємо хеш-функцію для рівномірного розподілу
    hashValue = hash(key)
    return abs(hashValue) mod numReducers

function customPartitioner(key: string, numReducers: integer): intege
r
    // Приклад: розподіл за першою літерою для текстових ключів
    if key.isEmpty() then
        return 0

    firstChar = key[0]
    bucketSize = 26 / numReducers  // для англійського алфавіту

    if isLetter(firstChar) then
        charIndex = toUpperCase(firstChar) - 'A'
        return min(charIndex / bucketSize, numReducers - 1)
    else
        return 0
```

Combiner - локальна оптимізація

Combiner виконує часткову агрегацію на стороні mapper'а:

```
function executeMapTaskWithCombiner(task: MapTask, combiner: function
): array[Partition]
    partitions = array[numReducers]

    // Використовуємо буфер для накопичення проміжних результатів
    buffer = HashMap<key, list[value]>()
    bufferSize = 0
    maxBufferSize = 10000  // налаштовуваний параметр

    for record in task.inputSplit do
        intermediateResults = task.mapFunction(record.key, record.val
ue)

        for (key2, value2) in intermediateResults do
            if key2 not in buffer then
                buffer[key2] = []
            buffer[key2].append(value2)
            bufferSize += 1

            // Коли буфер заповнений, застосовуємо combiner
            if bufferSize >= maxBufferSize then
                flushBufferWithCombiner(buffer, partitions, combiner)
                buffer.clear()
```

```
                bufferSize = 0

    // Обробляємо залишки в буфері
    if bufferSize > 0 then
        flushBufferWithCombiner(buffer, partitions, combiner)

    return partitions

function flushBufferWithCombiner(buffer: HashMap, partitions: array,
combiner: function)
    for (key, values) in buffer do
        // Застосовуємо combiner для локальної агрегації
        combinedValues = combiner(key, values)

        partitionId = partitioner(key, numReducers)
        for value in combinedValues do
            partitions[partitionId].add(key, value)
```

Shuffle фаза

Shuffle фаза відповідає за передачу даних від mapper'ів до reducer'ів:

```
function runShufflePhase(mapOutputs: array[array[Partition]], numRedu
cers: integer)
    // Для кожного reducer'а збираємо його партиції від всіх mapper'і
в
    reducerInputs = array[numReducers]

    for reducerId = 0 to numReducers - 1 do
        reducerInputs[reducerId] = ShuffledData()

        // Збираємо партиції від всіх mapper'ів
        for mapperId = 0 to mapOutputs.length - 1 do
            partition = fetchPartition(mapperId, reducerId)
            reducerInputs[reducerId].addPartition(partition)

        // Виконуємо злиття відсортованих партицій
        reducerInputs[reducerId].mergeSort()

    return reducerInputs

function fetchPartition(mapperId: integer, partitionId: integer): Par
tition
    // Завантажуємо партицію з диска або по мережі
    partitionPath = getPartitionPath(mapperId, partitionId)

    if isLocal(partitionPath) then
```

```
        return loadPartitionFromDisk(partitionPath)
    else
        return fetchPartitionOverNetwork(mapperId, partitionId)
```

Reduce фаза

Reduce фаза агрегує всі значення для кожного унікального ключа:

```
function runReducePhase(job: MapReduceJob, shuffledData: array[Shuffl
edData])
    reduceTasks = []

    // Створюємо reduce задачі
    for i = 0 to shuffledData.length - 1 do
        task = ReduceTask(
            taskId = i,
            partitionId = i,
            reduceFunction = job.reduceFunction,
            inputData = shuffledData[i]
        )
        reduceTasks.append(task)

    // Виконуємо reduce задачі паралельно
    results = parallelExecute(reduceTasks, executeReduceTask)
    return results

function executeReduceTask(task: ReduceTask): array[(key, value)]
    output = []

    // Групуємо значення за ключами
    currentKey = null
    currentValues = []

    for (key, value) in task.inputData do
        if key != currentKey then
            if currentKey != null then
                // Обробляємо попередню групу
                reducedValues = task.reduceFunction(currentKey, curre
ntValues)
                for resultValue in reducedValues do
                    output.append((currentKey, resultValue))

            currentKey = key
            currentValues = [value]
        else
            currentValues.append(value)
```

```
    // Обробляємо останню групу
    if currentKey != null then
        reducedValues = task.reduceFunction(currentKey, currentValues
)
        for resultValue in reducedValues do
            output.append((currentKey, resultValue))

    // Зберігаємо результати
    saveReduceOutput(task.taskId, output)
    return output
```

Обробка помилок та відмовостійкість

MapReduce забезпечує автоматичне відновлення після збоїв:

```
function executeMapReduceJobWithFaultTolerance(job: MapReduceJob)
    // Розділяємо вхідні дані
    inputSplits = splitInput(job.inputPath, job.numMappers)

    // Map фаза з повторними спробами
    mapResults = []
    for i = 0 to inputSplits.length - 1 do
        maxRetries = 3
        retryCount = 0
        success = false

        while not success and retryCount < maxRetries do
            try
                result = executeMapTask(createMapTask(i, inputSplits[
i]))
                mapResults.append(result)
                success = true
            catch Exception e
                retryCount += 1
                if retryCount < maxRetries then
                    // Перепризначаємо задачу іншому вузлу
                    reassignToHealthyNode(i)
                else
                    throw JobFailedException("Map task " + i + " fail
ed")

    // Shuffle фаза
    shuffledData = runShufflePhase(mapResults, job.numReducers)

    // Reduce фаза з повторними спробами
    reduceResults = []
    for i = 0 to job.numReducers - 1 do
```

```
        executeReduceTaskWithRetry(i, shuffledData[i], reduceResults)

    // Об'єднуємо результати
    mergeOutputFiles(reduceResults, job.outputPath)
```

Оптимізації MapReduce

Сучасні реалізації включають багато оптимізацій:

```
function optimizedMapReduce(job: MapReduceJob)
    // Спекулятивне виконання
    enableSpeculativeExecution(job)

    // Локальність даних
    scheduleTasksWithDataLocality(job)

    // Компресія проміжних даних
    enableIntermediateCompression(job)

    // Динамічне налаштування кількості reducer'ів
    job.numReducers = estimateOptimalReducers(job)

    // Виконання job'а
    executeMapReduceJob(job)

function enableSpeculativeExecution(job: MapReduceJob)
    // Відстежуємо прогрес кожної задачі
    taskProgressMonitor = TaskProgressMonitor()

    // Якщо задача виконується повільніше за середнє
    slowTaskThreshold = 1.5  // 150% від середнього часу

    taskProgressMonitor.onSlowTask = (taskId, taskType) => {
        if taskType == "map" then
            // Запускаємо дублюючу map задачу
            launchBackupMapTask(taskId)
        else
            // Запускаємо дублюючу reduce задачу
            launchBackupReduceTask(taskId)
    }

function scheduleTasksWithDataLocality(job: MapReduceJob)
    // Призначаємо задачі на вузли, де зберігаються дані
    for inputSplit in job.inputSplits do
        dataNodes = getNodesStoringData(inputSplit)

        // Пріоритет: той самий вузол > той самий rack > інший rack
```

```
        selectedNode = selectBestNode(dataNodes, availableNodes)
        scheduleMapTask(inputSplit, selectedNode)
```

Область застосування

MapReduce знайшов широке застосування в різних сферах обробки великих даних:

- Веб-індексація: Google використовує MapReduce для побудови пошукового індексу, обробляючи петабайти веб-сторінок.

- Аналіз логів: Обробка та аналіз величезних обсягів логів серверів для виявлення патернів та аномалій.

- Машинне навчання: Тренування моделей на великих датасетах, розподілені алгоритми кластеризації та класифікації.

- Обробка природної мови: Підрахунок n-грам, побудова мовних моделей, машинний переклад.

- Біоінформатика: Аналіз геномних послідовностей, вирівнювання ДНК, пошук патернів у біологічних даних.

- Фінансова аналітика: Розрахунок ризиків, аналіз транзакцій, виявлення шахрайства.

- Соціальні мережі: Аналіз графів друзів, рекомендаційні системи, виявлення спільнот.

Приклади застосування

Приклад 1: Підрахунок слів (Word Count)

Класичний приклад для демонстрації MapReduce:

```
// Вхідні дані
Input:
    file1: "hello world hello"
    file2: "world mapreduce hello"

// Map функція
function map(filename: string, content: string)
    for word in content.split(" ") do
        emit(word, 1)

// Map фаза виводить:
Mapper 1 (file1):
    ("hello", 1)
    ("world", 1)
    ("hello", 1)
```

```
Mapper 2 (file2):
    ("world", 1)
    ("mapreduce", 1)
    ("hello", 1)

// Shuffle фаза групує за ключами:
Reducer input:
    ("hello", [1, 1, 1])
    ("mapreduce", [1])
    ("world", [1, 1])

// Reduce функція
function reduce(word: string, counts: list[integer])
    sum = 0
    for count in counts do
        sum += count
    emit(word, sum)

// Reduce фаза виводить:
Output:
    ("hello", 3)
    ("mapreduce", 1)
    ("world", 2)
```

Приклад 2: Обчислення середньої температури

Аналіз метеорологічних даних:

```
// Вхідні дані (місто, дата, температура)
Input:
    "Київ,2024-01-01,2.5"
    "Львів,2024-01-01,-1.2"
    "Київ,2024-01-02,3.1"
    "Львів,2024-01-02,-0.5"
    "Київ,2024-01-03,1.8"

// Map функція
function map(lineNumber: integer, line: string)
    parts = line.split(",")
    city = parts[0]
    temperature = parseFloat(parts[2])
    emit(city, temperature)

// Map вивід:
    ("Київ", 2.5)
    ("Львів", -1.2)
```

```
    ("Київ", 3.1)
    ("Львів", -0.5)
    ("Київ", 1.8)

// Shuffle групує:
    ("Київ", [2.5, 3.1, 1.8])
    ("Львів", [-1.2, -0.5])

// Reduce функція
function reduce(city: string, temperatures: list[float])
    sum = 0.0
    count = 0
    for temp in temperatures do
        sum += temp
        count += 1
    average = sum / count
    emit(city, average)

// Output:
    ("Київ", 2.47)
    ("Львів", -0.85)
```

Приклад 3: Інвертований індекс

Побудова інвертованого індексу для пошукової системи:

```
// Вхідні документи
Input:
    doc1: "MapReduce is a programming model"
    doc2: "The model is simple but powerful"
    doc3: "MapReduce programming is distributed"

// Map функція
function map(docId: string, content: string)
    words = content.toLowerCase().split(" ")
    for word in words do
        emit(word, docId)

// Map вивід:
    ("mapreduce", "doc1")
    ("is", "doc1")
    ("a", "doc1")
    ("programming", "doc1")
    ("model", "doc1")
    ("the", "doc2")
    ("model", "doc2")
    ("is", "doc2")
```

```
    ("simple", "doc2")
    ("but", "doc2")
    ("powerful", "doc2")
    ("mapreduce", "doc3")
    ("programming", "doc3")
    ("is", "doc3")
    ("distributed", "doc3")

// Після shuffle:
    ("a", ["doc1"])
    ("but", ["doc2"])
    ("distributed", ["doc3"])
    ("is", ["doc1", "doc2", "doc3"])
    ("mapreduce", ["doc1", "doc3"])
    ("model", ["doc1", "doc2"])
    ("powerful", ["doc2"])
    ("programming", ["doc1", "doc3"])
    ("simple", ["doc2"])
    ("the", ["doc2"])

// Reduce функція
function reduce(word: string, docIds: list[string])
    // Видаляємо дублікати та сортуємо
    uniqueDocs = removeDuplicates(docIds)
    sortedDocs = sort(uniqueDocs)
    emit(word, joinWithComma(sortedDocs))

// Output:
    ("a", "doc1")
    ("but", "doc2")
    ("distributed", "doc3")
    ("is", "doc1,doc2,doc3")
    ("mapreduce", "doc1,doc3")
    ("model", "doc1,doc2")
    ("powerful", "doc2")
    ("programming", "doc1,doc3")
    ("simple", "doc2")
    ("the", "doc2")
```

Приклад 4: Join операція

Об’єднання двох датасетів:

```
// Датасет 1: Користувачі (userId, name)
users:
    "1,Олександр"
    "2,Марія"
```

```
    "3,Петро"

// Датасет 2: Замовлення (orderId, userId, amount)
orders:
    "101,1,250.00"
    "102,2,180.50"
    "103,1,320.00"
    "104,3,150.00"

// Map функція для обох датасетів
function map(source: string, key: string, value: string)
    if source == "users" then
        parts = value.split(",")
        userId = parts[0]
        name = parts[1]
        emit(userId, ("user", name))
    else if source == "orders" then
        parts = value.split(",")
        orderId = parts[0]
        userId = parts[1]
        amount = parts[2]
        emit(userId, ("order", orderId + "," + amount))

// Map вивід:
    ("1", ("user", "Олександр"))
    ("2", ("user", "Марія"))
    ("3", ("user", "Петро"))
    ("1", ("order", "101,250.00"))
    ("2", ("order", "102,180.50"))
    ("1", ("order", "103,320.00"))
    ("3", ("order", "104,150.00"))

// Після shuffle:
    ("1", [("user", "Олександр"), ("order", "101,250.00"), ("order",
"103,320.00")])
    ("2", [("user", "Марія"), ("order", "102,180.50")])
    ("3", [("user", "Петро"), ("order", "104,150.00")])

// Reduce функція
function reduce(userId: string, values: list[(type, data)])
    userName = null
    orders = []

    for (type, data) in values do
        if type == "user" then
```

```
            userName = data
        else if type == "order" then
            orders.append(data)

    if userName != null then
        for orderData in orders do
            emit(userId, userName + "," + orderData)

// Output:
    ("1", "Олександр,101,250.00")
    ("1", "Олександр,103,320.00")
    ("2", "Марія,102,180.50")
    ("3", "Петро,104,150.00")
```

Приклад 5: PageRank алгоритм

Спрощена версія PageRank з використанням MapReduce:

```
// Вхідні дані: граф у форматі (сторінка, [список посилань], поточний
rank)
Input:
    "A,[B,C],1.0"
    "B,[C],1.0"
    "C,[A],1.0"

// Map функція для однієї ітерації PageRank
function map(key: string, value: string)
    parts = parsePageData(value)
    page = parts.page
    links = parts.links
    rank = parts.rank

    numLinks = links.length
    if numLinks > 0 then
        contributedRank = rank / numLinks

        // Розподіляємо rank між всіма посиланнями
        for targetPage in links do
            emit(targetPage, ("rank", contributedRank))

    // Передаємо структуру графа
    emit(page, ("links", joinLinks(links)))

// Map вивід:
    ("B", ("rank", 0.5))    // від A
    ("C", ("rank", 0.5))    // від A
    ("A", ("links", "B,C"))
```

```
    ("C", ("rank", 1.0))    // від B
    ("B", ("links", "C"))
    ("A", ("rank", 1.0))    // від C
    ("C", ("links", "A"))

// Reduce функція
function reduce(page: string, values: list[(type, data)])
    links = ""
    rankSum = 0.0
    dampingFactor = 0.85

    for (type, data) in values do
        if type == "rank" then
            rankSum += parseFloat(data)
        else if type == "links" then
            links = data

    // Формула PageRank
    newRank = (1 - dampingFactor) + dampingFactor * rankSum
    emit(page, page + ",[" + links + "]," + newRank)

// Output після однієї ітерації:
    ("A", "A,[B,C],1.0")     // (1-0.85) + 0.85*1.0 = 1.0
    ("B", "B,[C],0.575")     // (1-0.85) + 0.85*0.5 = 0.575
    ("C", "C,[A],1.425")     // (1-0.85) + 0.85*1.5 = 1.425
```

Переваги та недоліки

Переваги

- Простота програмування: Розробник пише лише дві функції (map та reduce), а система автоматично обробляє паралелізацію, розподіл даних та відмовостійкість.

- Масштабованість: Лінійно масштабується від одного комп’ютера до тисяч вузлів без зміни коду.

- Відмовостійкість: Автоматичне відновлення після збоїв окремих вузлів через реплікацію даних та перезапуск задач.

- Локальність даних: Обчислення виконуються там, де зберігаються дані, мінімізуючи мережевий трафік.

- Гнучкість: Підходить для широкого спектру задач - від простого підрахунку до складного машинного навчання.

- Оптимізації: Вбудовані оптимізації як combiner, спекулятивне виконання, компресія.

Недоліки

- Неефективність для ітеративних алгоритмів: Кожна ітерація вимагає повного циклу MapReduce з записом на диск.

- Затримки: Високі накладні витрати на запуск задач роблять MapReduce непридатним для обробки в реальному часі.

- Обмежена модель обчислень: Не всі алгоритми природно вписуються в map-reduce парадигму.

- Використання диску: Проміжні результати зберігаються на диску, що сповільнює обробку.

- Складність налагодження: Розподілена природа ускладнює відлагодження та профілювання.

- Неефективність для малих даних: Накладні витрати перевищують користь для датасетів менше кількох гігабайтів.

Контрольні запитання

1. Що таке MapReduce та які основні фази обробки даних в цій парадигмі?

2. Поясніть роль функції Partitioner та як вона впливає на розподіл даних між reducer'ами.

3. Як працює механізм відмовостійкості в MapReduce?

4. Що таке Combiner і в яких випадках його використання дає найбільший ефект?

5. Які типи задач найкраще підходять для MapReduce, а які - ні?

6. Як забезпечується локальність даних при виконанні map задач?

7. Поясніть процес shuffle фази та її важливість для коректної роботи.

8. Що таке спекулятивне виконання і як воно покращує продуктивність?

9. Чому MapReduce неефективний для ітеративних алгоритмів?

10. Як MapReduce масштабується при збільшенні обсягу даних та кількості вузлів?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження парадигми MapReduce

2. Мета

Вивчити принципи розподіленої обробки даних за допомогою MapReduce, реалізувати власну спрощену версію фреймворку, дослідити ефективність паралельної обробки та масштабованість алгоритмів.

3. Завдання

Реалізувати локальну версію MapReduce фреймворку з підтримкою основних функцій. Розробити кілька MapReduce програм для різних задач. Провести експерименти з продуктивністю та масштабованістю.

4. Опис

Лабораторна робота передбачає створення спрощеної версії MapReduce, що включає планувальник задач, виконавців для map та reduce фаз, механізм shuffle. Студенти реалізують класичні алгоритми та досліджують їх поведінку при різних розмірах даних та кількості воркерів.

5. Порядок виконання

- Реалізувати базові класи для MapReduceJob, Mapper, Reducer
- Створити планувальник для розподілу задач між воркерами
- Реалізувати механізм розбиття вхідних даних на частини
- Розробити shuffle механізм для групування проміжних результатів
- Додати підтримку Combiner для оптимізації
- Реалізувати обробку помилок та перезапуск задач
- Створити програми WordCount, InvertedIndex, PageRank
- Додати можливість візуалізації виконання job’ів
- Провести бенчмарки для різних конфігурацій
- Порівняти з послідовною версією алгоритмів

6. Результат

Працюючий MapReduce фреймворк з прикладами використання, графіки залежності часу виконання від кількості воркерів та розміру даних, аналіз ефективності розпаралелювання, рекомендації щодо оптимального використання MapReduce.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Базові класи фреймворку (Job, Mapper, Reducer, Partitioner)
- Планувальник задач з підтримкою локальності даних

- Механізм shuffle з сортуванням та групуванням
- Реалізації алгоритмів (WordCount, InvertedIndex, тощо)
- Система моніторингу виконання

2. Продемонструвати роботу додатку:
- Запуск WordCount на текстових файлах різного розміру
- Побудова інвертованого індексу для колекції документів
- Виконання однієї ітерації PageRank
- Візуалізація розподілу задач між воркерами
- Демонстрація відновлення після збою воркера

3. Бути готовим відповісти на питання:
- Як забезпечується рівномірний розподіл навантаження?
- Чому важливе локальне виконання map задач?
- Як оптимізувати shuffle фазу?
- Коли використання Combiner дає найбільший ефект?

Тести

1. Яка функція виконується першою в MapReduce?
- 1) Reduce
- 2) Map
- 3) Shuffle
- 4) Combine

2. Що відбувається під час Shuffle фази?
- 1) Сортування вхідних даних
- 2) Групування проміжних результатів за ключами
- 3) Виконання reduce функції
- 4) Розподіл даних між mapper’ами

3. Для чого використовується Combiner?
- 1) Об’єднання вхідних файлів
- 2) Локальна агрегація на стороні mapper’а
- 3) Фінальне об’єднання результатів
- 4) Розподіл задач

4. Яка складність сортування в MapReduce при M мапперах та R редюсерах?

- 1) O(N)

- 2) O(N log N)

- 3) O(N/M × log(N/M))

- 4) $O(N^2)$

5. Що визначає Partitioner?

- 1) Розмір input split

- 2) Який reducer отримає конкретний ключ

- 3) Кількість mapper’ів

- 4) Формат вихідних даних

Відповіді:

1-2,

2-2,

3-2,

4-3,

5-2

Тести підвищеної складності

1. Чому MapReduce використовує сортування в shuffle фазі?

- 1) Для прискорення обробки

- 2) Щоб reduce отримував відсортовані дані та міг ефективно групувати

- 3) Це вимога файлової системи

- 4) Для зменшення пам’яті

2. Як спекулятивне виконання покращує продуктивність?

- 1) Прискорює всі задачі

- 2) Дублює повільні задачі на інших вузлах

- 3) Використовує більше пам’яті

- 4) Зменшує кількість mapper’ів

3. Яка основна проблема використання MapReduce для ітеративних алгоритмів?

- 1) Неможливість реалізації

- 2) Накладні витрати на запис/читання між ітераціями
- 3) Обмеження пам’яті
- 4) Складність програмування

4. Коли Combiner НЕ можна використовувати?
- 1) При підрахунку суми
- 2) При знаходженні максимуму
- 3) При обчисленні середнього арифметичного
- 4) При підрахунку кількості

5. Що таке “data skew” в контексті MapReduce?
- 1) Пошкоджені дані
- 2) Нерівномірний розподіл даних між reducer’ами
- 3) Помилка в map функції
- 4) Втрата даних

6. Як MapReduce забезпечує exactly-once семантику?
- 1) Використовує транзакції
- 2) Атомарний запис результатів та детерміністичні функції
- 3) Блокує паралельне виконання
- 4) Не забезпечує

7. Чому secondary sort важливий для деяких алгоритмів?
- 1) Прискорює виконання
- 2) Дозволяє контролювати порядок значень в reduce
- 3) Зменшує використання пам’яті
- 4) Спрощує код

8. Яка типова причина OutOfMemoryError в reducer’і?
- 1) Занадто багато reducer’ів
- 2) Один ключ має занадто багато значень
- 3) Повільна мережа
- 4) Малий input split

9. Як in-mapper combining відрізняється від звичайного Combiner?
- 1) Виконується після map

- 2) Агрегація відбувається в пам’яті mapper’а

- 3) Працює повільніше

- 4) Не відрізняється

10. Чому MapReduce погано підходить для обробки графів?

- 1) Графи занадто складні

- 2) Потрібно багато ітерацій з повним перезаписом стану

- 3) Немає підтримки рекурсії

- 4) Обмеження на розмір ключів

Відповіді:

1. 2) Сортування дозволяє ефективно групувати всі значення одного ключа

2. 2) Повільні задачі можуть затримувати весь job, дублювання дозволяє завершити швидше

3. 2) Кожна ітерація вимагає повного циклу з записом на диск

4. 3) Для середнього потрібно знати кількість та суму, що ускладнює часткову агрегацію

5. 2) Коли один reducer отримує набагато більше даних, він стає вузьким місцем

6. 2) Детерміністичність та атомарність запису гарантують однократне виконання

7. 2) Дозволяє обробляти значення в певному порядку, наприклад, за часом

8. 2) Всі значення одного ключа завантажуються в пам’ять одночасно

9. 2) Зменшує кількість проміжних записів через агрегацію в пам’яті

10. 2) Алгоритми на графах вимагають багато ітерацій з повним перечитуванням

Завдання для самостійної роботи

1. Реалізуйте алгоритм k-means кластеризації використовуючи MapReduce парадигму.

2. Створіть MapReduce програму для обчислення TF-IDF для колекції документів.

3. Розробіть distributed grep - пошук рядків за регулярним виразом у великих логах.

4. Реалізуйте обчислення матричного множення за допомогою MapReduce.

5. Створіть програму для знаходження трикутників у графі (triangle counting).

6. Розробіть MapReduce job для об’єднання кількох відсортованих файлів.

7. Реалізуйте distributed sort з використанням sampling для вибору розділювачів.

8. Створіть систему дедуплікації великих датасетів через MapReduce.

9. Розробіть алгоритм виявлення аномалій у часових рядах.

10. Реалізуйте SQL-подібні операції (GROUP BY, JOIN) через MapReduce.

Теми лабораторних робіт

1. Базова реалізація MapReduce фреймворку та класичних алгоритмів.

2. Оптимізація MapReduce для SSD накопичувачів з урахуванням їх особливостей.

3. Порівняння MapReduce з Apache Spark для ітеративних алгоритмів.

4. Реалізація відмовостійкого MapReduce з Byzantine fault tolerance.

5. Створення візуального дебагера для MapReduce програм.

6. Розробка auto-tuning системи для оптимізації параметрів MapReduce.

7. Інтеграція MapReduce з системами потокової обробки даних.

8. Реалізація MapReduce поверх розподіленої файлової системи.

9. Створення DSL для спрощення написання MapReduce програм.

10. Дослідження енергоефективності різних стратегій планування в MapReduce.

Розділ 14: Алгоритми для великих даних та потоків

Перейдемо до задач, в яких дані занадто великі, щоб помістится в оперативну пам’ять, і надходять так швидко, що їх неможливо зберегти повністю. Класичні алгоритми, які ми вивчали досі, часто базуються на припущенні, що вся необхідна інформація доступна в RAM для довільного доступу.

Однак у сучасних системах — від соціальних мереж та інтернет-сенсорів до фінансових транзакцій і наукових експериментів — це припущення більше не виконується.

Цей розділ присвячений сучасним підходам до обробки даних в умовах жорстких обмежень на пам’ять та швидкість. Ми розглянемо чотири фундаментальні стратегії, які дозволяють ефективно працювати з великими даними (Big Data) та потоками даних (Data Streams).

1. Наближені відповіді за допомогою sketch-структур: Що робити, коли точна відповідь вимагає занадто багато ресурсів? Ми познайомимося зі структурами, як-от Count-Min Sketch та HyperLogLog, які дозволяють отримати дуже точні наближені оцінки (наприклад, частоти елементів або кількості унікальних значень), використовуючи лише крихітну частину пам’яті.

2. Обробка нескінченних потоків: Коли дані надходять неперервним потоком і їх не можна зберегти, нам потрібні алгоритми, що працюють “на льоту”. Ми вивчимо Lossy Counting та Space-Saving, які дозволяють ідентифікувати найважливіші елементи в потоці, миттєво відкидаючи менш значущі.

3. Робота із зовнішньою пам’яттю: Якщо дані зберігаються на диску, вартість доступу до них стає головним вузьким місцем. Ми розглянемо LSM-дерева, що лежать в основі багатьох сучасних баз даних, та зовнішнє сортування злиттям — методи, оптимізовані для мінімізації повільних операцій читання/запису на диск.

4. Оптимізація для ієрархії пам’яті: Навіть коли дані в RAM, швидкість доступу до них нерівномірна (регістри, L1/L2/L3-кеші, основна пам’ять). Cache-oblivious алгоритми проєктуються так, щоб ефективно використовувати ієрархію пам’яті автоматично, не вимагаючи знань про конкретну архітектуру кешу.

Цей розділ озброїть вас інструментами для розробки ефективних рішень у реаліях сучасного інформаційного світу, де обсяг і швидкість даних є головними викликами.

14.1. Sketch-структури

Sketch-структури — це спеціальні, дуже компактні структури даних, які дозволяють отримувати наближені відповіді на запити про великі обсяги даних, використовуючи при цьому мінімальний обсяг пам'яті.

Вони є ключовим інструментом для роботи з потоками даних (data streams) та величезними наборами даних (Big Data), коли точний розрахунок є або неможливим через брак пам'яті, або занадто повільним.

Головна ідея sketch-структур полягає в тому, щоб пожертвувати точністю заради ефективності. Замість того, щоб зберігати всі дані, вони зберігають лише їхній стислий "відбиток" або "нарис" (англ. *sketch*). Цей нарис дозволяє швидко відповідати на певні типи запитів із гарантованою похибкою.

Ключові властивості:

- Компактність: Вони займають значно менше місця, ніж вихідні дані. Розмір sketch-структури, як правило, є фіксованим і не залежить від кількості оброблених елементів.

- Швидкість: Оновлення структури (додавання нового елемента) та виконання запиту відбуваються дуже швидко, зазвичай за постійний час.

- Наближеність: Результати не є точними, але мають математично доведені гарантії точності. Наприклад, відповідь може бути точною з імовірністю 99% або відхилятися від правильного значення не більше ніж на 1%.

Count-Min Sketch

Це структура для наближеного підрахунку частоти елементів у потоці.

- Задача: Скільки разів у потоці зустрічався елемент "A"?

- Як працює: Використовує кілька хеш-функцій для "розмазування" підрахунків по невеликій таблиці (матриці). Щоб отримати частоту елемента, структура хешує його і повертає мінімальне зі значень у відповідних комірках. Це допомагає зменшити вплив колізій (коли різні елементи потрапляють в одну комірку).

- Результат: Частота завжди буде не меншою за реальну, але похибка буде невеликою.

HyperLogLog (HLL)

Це структура для наближеного підрахунку кількості унікальних елементів у наборі даних (кардинальності).

- Задача: Скільки унікальних користувачів відвідало сайт за день, якщо логів мільярди?

- Як працює: Хешує кожен елемент і аналізує патерни в бітовому представленні хешів (зокрема, кількість нулів на початку). Інтуїтивно, якщо ви бачили хеш, що починається з великої кількості нулів, ви, ймовірно, перебрали багато різних елементів.

- Результат: Дозволяє оцінити кількість унікальних елементів до мільярдів, використовуючи лише кілька кілобайтів пам'яті з дуже високою точністю (наприклад, похибка ~2%).

Отже, sketch-структури є незамінними, коли вам потрібно аналізувати величезні обсяги даних "на льоту" і ви готові прийняти невелику, контрольовану неточність в обмін на колосальну економію ресурсів.

Sketch-структури: Count-Min

Теоретичні відомості

Count-Min Sketch є імовірнісною структурою даних для підрахунку частот елементів у потоках даних з використанням сублінійної пам'яті. Розроблена Гремом Кормоде та С. Мутукрішнаном у 2005 році, ця структура дозволяє ефективно оцінювати частоти елементів, використовуючи значно менше пам'яті, ніж потрібно для точного підрахунку, за рахунок невеликої контрольованої похибки.

Count-Min Sketch базується на використанні двовимірного масиву лічильників та набору незалежних хеш-функцій. Основна ідея полягає в тому, що кожен елемент хешується кількома незалежними хеш-функціями, і відповідні лічильники збільшуються. При запиті частоти елемента повертається мінімум серед усіх відповідних лічильників.

```
struct CountMinSketch {
    width: integer          // ширина масиву (w)
    depth: integer          // глибина масиву (d)
    count: array[d][w] of integer  // масив лічильників
```

```
    hashFunctions: array[d] of HashFunction  // незалежні хеш-функції
    totalCount: integer     // загальна кількість елементів
}
```

Розміри структури визначаються на основі бажаних параметрів точності:

- Ширина: $w = \lceil e/\epsilon \rceil$

- Глибина: $d = \lceil \ln(1/\delta) \rceil$

де: - ϵ - відносна похибка (наприклад, 0.01 для 1% похибки) - δ - ймовірність перевищення похибки (наприклад, 0.05 для 95% впевненості) - e - основа натурального логарифма (≈ 2.718)

Ініціалізація структури

Процес створення Count-Min Sketch включає ініціалізацію масиву та генерацію хеш-функцій:

```
function initCountMinSketch(epsilon: float, delta: float): CountMinSk
etch
    // Обчислюємо розміри на основі параметрів точності
    width = ceil(e / epsilon)
    depth = ceil(ln(1 / delta))

    // Ініціалізуємо масив лічильників нулями
    count = array[depth][width]
    for i = 0 to depth - 1 do
        for j = 0 to width - 1 do
            count[i][j] = 0

    // Генеруємо незалежні хеш-функції
    hashFunctions = array[depth]
    for i = 0 to depth - 1 do
        hashFunctions[i] = generateHashFunction(i, width)

    cms = CountMinSketch(
        width = width,
        depth = depth,
        count = count,
        hashFunctions = hashFunctions,
        totalCount = 0
    )

    return cms
```

Генерація хеш-функцій

Для забезпечення незалежності хеш-функцій часто використовується універсальне хешування:

```
function generateHashFunction(seed: integer, width: integer): HashFun
ction
    // Параметри для універсального хешування
    prime = nextPrime(width * 100)  // велике просте число
    a = randomInt(1, prime - 1)
    b = randomInt(0, prime - 1)

    // Повертаємо функцію, що реалізує (a*x + b) mod prime mod width
    return function(x: any): integer
        hashValue = hash(x)  // базова хеш-функція (наприклад, Murmur
Hash)
        return ((a * hashValue + b) mod prime) mod width
```

Операція UPDATE (додавання елемента)

Операція оновлення збільшує лічильники для заданого елемента:

```
function update(cms: CountMinSketch, item: any, count: integer = 1)
    // Оновлюємо загальну кількість
    cms.totalCount += count

    // Для кожної хеш-функції
    for i = 0 to cms.depth - 1 do
        // Обчислюємо позицію в i-му рядку
        j = cms.hashFunctions[i](item)

        // Збільшуємо відповідний лічильник
        cms.count[i][j] += count
```

Операція QUERY (запит частоти)

Запит повертає оцінку частоти елемента:

```
function query(cms: CountMinSketch, item: any): integer
    // Ініціалізуємо мінімум максимальним значенням
    minCount = INFINITY

    // Перевіряємо всі рядки
    for i = 0 to cms.depth - 1 do
        // Знаходимо позицію в i-му рядку
        j = cms.hashFunctions[i](item)

        // Оновлюємо мінімум
        if cms.count[i][j] < minCount then
            minCount = cms.count[i][j]
```

```
    return minCount
```

Операція MERGE (об’єднання sketch’ів)

Count-Min Sketch підтримує ефективне об’єднання кількох структур:

```
function merge(cms1: CountMinSketch, cms2: CountMinSketch): CountMinS
ketch
    // Перевіряємо сумісність розмірів
    if cms1.width != cms2.width or cms1.depth != cms2.depth then
        throw Error("Incompatible sketch dimensions")

    // Створюємо новий sketch з тими ж параметрами
    result = CountMinSketch(
        width = cms1.width,
        depth = cms1.depth,
        count = array[cms1.depth][cms1.width],
        hashFunctions = cms1.hashFunctions,  // використовуємо ті ж х
еш-функції
        totalCount = cms1.totalCount + cms2.totalCount
    )

    // Поелементно додаємо лічильники
    for i = 0 to result.depth - 1 do
        for j = 0 to result.width - 1 do
            result.count[i][j] = cms1.count[i][j] + cms2.count[i][j]

    return result
```

Розширені операції

1. Inner Product (скалярний добуток)

Count-Min Sketch може оцінювати скалярний добуток двох потоків:

```
function innerProduct(cms1: CountMinSketch, cms2: CountMinSketch): in
teger
    // Перевіряємо сумісність
    if cms1.width != cms2.width or cms1.depth != cms2.depth then
        throw Error("Incompatible sketches")

    minProduct = INFINITY

    // Для кожного рядка обчислюємо скалярний добуток
    for i = 0 to cms1.depth - 1 do
        rowProduct = 0
        for j = 0 to cms1.width - 1 do
            rowProduct += cms1.count[i][j] * cms2.count[i][j]
```

```
        if rowProduct < minProduct then
            minProduct = rowProduct

    return minProduct
```

2. Range Query (запит діапазону)

Для підтримки запитів діапазону можна використовувати ієрархічну структуру:

```
struct HierarchicalCountMinSketch {
    levels: array of CountMinSketch  // sketch для кожного рівня
    maxValue: integer                // максимальне значення ключа
}

function rangeQuery(hcms: HierarchicalCountMinSketch, start: integer,
end: integer): integer
    sum = 0
    currentStart = start

    while currentStart <= end do
        // Знаходимо найбільший блок, що поміщається в діапазон
        level = 0
        blockSize = 1

        while level < hcms.levels.length - 1 do
            nextBlockSize = blockSize * 2
            blockStart = (currentStart / nextBlockSize) * nextBlockSi
ze

            if blockStart == currentStart and blockStart + nextBlockS
ize - 1 <= end then
                level += 1
                blockSize = nextBlockSize
            else
                break

        // Додаємо частоту блоку
        blockId = currentStart / blockSize
        sum += query(hcms.levels[level], blockId)

        currentStart += blockSize

    return sum
```

Оптимізації та варіації

1. Conservative Update

Консервативне оновлення зменшує переоцінку:

```
function conservativeUpdate(cms: CountMinSketch, item: any, count: in
teger = 1)
    // Спочатку знаходимо поточну оцінку
    currentEstimate = query(cms, item)

    // Оновлюємо тільки мінімальні лічильники
    for i = 0 to cms.depth - 1 do
        j = cms.hashFunctions[i](item)

        if cms.count[i][j] == currentEstimate then
            cms.count[i][j] += count
```

2. Count-Min-Log Sketch

Для дуже великих частот використовується логарифмічне представлення:

```
struct CountMinLogSketch {
    width: integer
    depth: integer
    count: array[depth][width] of byte  // зберігаємо логарифми
    base: float  // основа логарифма
}

function updateLog(cmls: CountMinLogSketch, item: any)
    for i = 0 to cmls.depth - 1 do
        j = cmls.hashFunctions[i](item)

        // Ймовірнісне збільшення
        currentValue = cmls.base ^ cmls.count[i][j]
        probability = 1.0 / currentValue

        if random() < probability then
            cmls.count[i][j] += 1

function queryLog(cmls: CountMinLogSketch, item: any): float
    minLog = INFINITY

    for i = 0 to cmls.depth - 1 do
        j = cmls.hashFunctions[i](item)
        if cmls.count[i][j] < minLog then
            minLog = cmls.count[i][j]

    return cmls.base ^ minLog
```

Аналіз похибок

Count-Min Sketch гарантує, що оцінка частоти ніколи не буде менша за справжню частоту:

$$\hat{f}_i \geq f_i$$

де $\hat{f}_i$ - оцінка частоти елемента i, а f_i - справжня частота.

З ймовірністю принаймні $1 - \delta$:

$$\hat{f}_i \leq f_i + \epsilon \cdot N$$

де N - загальна кількість елементів у потоці.

Паралельна обробка

Count-Min Sketch природно підтримує паралельну обробку:

```
function parallelUpdate(cms: CountMinSketch, items: array of any, num
Threads: integer)
    // Розділяємо елементи між потоками
    chunksSize = ceil(items.length / numThreads)
    threads = array[numThreads]

    for t = 0 to numThreads - 1 do
        startIdx = t * chunksSize
        endIdx = min((t + 1) * chunksSize, items.length)

        threads[t] = createThread(() => {
            // Кожен потік оновлює свій діапазон рядків
            for item in items[startIdx:endIdx] do
                for i = 0 to cms.depth - 1 do
                    j = cms.hashFunctions[i](item)

                    // Атомарне збільшення
                    atomicIncrement(cms.count[i][j])
        })

    // Чекаємо завершення всіх потоків
    for thread in threads do
        thread.join()
```

Область застосування

Count-Min Sketch широко використовується в системах обробки великих потоків даних:

- Мережевий моніторинг: Підрахунок трафіку для IP-адрес, виявлення DDoS-атак, аналіз мережевих потоків.

- Бази даних: Оцінка селективності запитів, підрахунок кардинальності, оптимізація планів виконання.

- Обробка природної мови: Підрахунок частот слів у великих корпусах тексту, виявлення трендових тем.

- Рекомендаційні системи: Відстеження популярності товарів, підрахунок взаємодій користувачів.

- Потокова аналітика: Обробка логів в реальному часі, моніторинг метрик, виявлення аномалій.

- Розподілені системи: Агрегація статистики з багатьох вузлів, злиття частотних розподілів.

Приклади застосування

Приклад 1: Базовий підрахунок частот

Розглянемо потік символів та підрахунок їх частот:

```
Потік: A, B, A, C, B, A, D, A, B, C

Параметри: ε = 0.3, δ = 0.1
width = ceil(e / 0.3) = 10
depth = ceil(ln(1 / 0.1)) = 3

Ініціалізація:
count = [[0,0,0,0,0,0,0,0,0,0],
         [0,0,0,0,0,0,0,0,0,0],
         [0,0,0,0,0,0,0,0,0,0]]

Хеш-функції:
h₁(x) = (3x + 1) mod 10
h₂(x) = (5x + 2) mod 10
h₃(x) = (7x + 3) mod 10

Припустимо: hash(A)=1, hash(B)=2, hash(C)=3, hash(D)=4

Обробка потоку:
1. A: h₁(1)=4, h₂(1)=7, h₃(1)=0
   count[0][4]++, count[1][7]++, count[2][0]++

2. B: h₁(2)=7, h₂(2)=2, h₃(2)=7
   count[0][7]++, count[1][2]++, count[2][7]++

... (продовження для всіх елементів)
```

```
Фінальний стан:
count = [[0,0,0,1,4,0,0,3,0,0],
         [0,0,3,0,0,0,0,4,1,0],
         [4,0,0,0,0,0,0,3,0,1]]

Запити:
query(A) = min(count[0][4], count[1][7], count[2][0]) = min(4,4,4) =
4 ✓
query(B) = min(count[0][7], count[1][2], count[2][7]) = min(3,3,3) =
3 ✓
query(C) = min(count[0][0], count[1][5], count[2][3]) = min(0,0,0) =
2 ✓
query(D) = min(count[0][3], count[1][0], count[2][8]) = min(1,0,0) =
1 ✓
```

Приклад 2: Виявлення heavy hitters

Знаходження елементів з частотою вище порогу:

```
function findHeavyHitters(cms: CountMinSketch, threshold: float): set
of items
    heavyHitters = set()
    minHeapSize = 1 / threshold
    minHeap = MinHeap(maxSize = minHeapSize)

    // Зберігаємо потенційні heavy hitters під час оновлення
    function updateWithTracking(item: any, count: integer = 1)
        update(cms, item, count)

        estimatedCount = query(cms, item)
        if estimatedCount >= threshold * cms.totalCount then
            minHeap.insert(item, estimatedCount)

            if minHeap.size() > minHeapSize then
                minHeap.extractMin()

    // Верифікуємо кандидатів
    candidates = minHeap.getAllItems()
    for candidate in candidates do
        if query(cms, candidate) >= threshold * cms.totalCount then
            heavyHitters.add(candidate)

    return heavyHitters

// Приклад використання
stream = ["apple", "banana", "apple", "cherry", "apple", "banana", ..
.]
```

```
threshold = 0.2  // елементи з частотою > 20%

cms = initCountMinSketch(0.01, 0.05)
for item in stream do
    updateWithTracking(item, 1)

heavyHitters = findHeavyHitters(cms, threshold)
// Результат: {"apple", "banana"} якщо їх частоти > 20%
```

Приклад 3: Мережевий моніторинг

Відстеження top-k IP-адрес за трафіком:

```
struct NetworkMonitor {
    cms: CountMinSketch
    topK: MinHeap
    k: integer
}

function processPacket(monitor: NetworkMonitor, packet: NetworkPacket
)
    srcIP = packet.sourceIP
    packetSize = packet.size

    // Оновлюємо sketch
    update(monitor.cms, srcIP, packetSize)

    // Отримуємо оцінку трафіку для цієї IP
    estimatedTraffic = query(monitor.cms, srcIP)

    // Оновлюємо top-k
    if monitor.topK.contains(srcIP) then
        monitor.topK.updateKey(srcIP, estimatedTraffic)
    else if monitor.topK.size() < monitor.k then
        monitor.topK.insert(srcIP, estimatedTraffic)
    else if estimatedTraffic > monitor.topK.getMin().traffic then
        monitor.topK.extractMin()
        monitor.topK.insert(srcIP, estimatedTraffic)

// Приклад роботи
monitor = NetworkMonitor(
    cms = initCountMinSketch(0.001, 0.01),
    topK = MinHeap(),
    k = 100
)

// Обробка пакетів
```

```
packets = [
    {sourceIP: "192.168.1.1", size: 1500},
    {sourceIP: "10.0.0.1", size: 800},
    {sourceIP: "192.168.1.1", size: 1200},
    ...
]

for packet in packets do
    processPacket(monitor, packet)

// Отримання top-100 IP за трафіком
topIPs = monitor.topK.getAllItems()
```

Приклад 4: Підрахунок унікальних елементів

Комбінування з HyperLogLog для оцінки кардинальності:

```
struct CountMinHLL {
    cms: CountMinSketch
    hlls: array[depth][width] of HyperLogLog
}

function updateUnique(cmhll: CountMinHLL, item: any, value: any)
    for i = 0 to cmhll.cms.depth - 1 do
        j = cmhll.cms.hashFunctions[i](item)

        // Оновлюємо Count-Min
        cmhll.cms.count[i][j] += 1

        // Додаємо до HyperLogLog
        cmhll.hlls[i][j].add(value)

function queryUnique(cmhll: CountMinHLL, item: any): integer
    minCardinality = INFINITY

    for i = 0 to cmhll.cms.depth - 1 do
        j = cmhll.cms.hashFunctions[i](item)

        cardinality = cmhll.hlls[i][j].estimate()
        if cardinality < minCardinality then
            minCardinality = cardinality

    return minCardinality

// Приклад: підрахунок унікальних відвідувачів сайту
cmhll = initCountMinHLL(0.01, 0.05)
```

```
visits = [
    {page: "/home", userId: "user1"},
    {page: "/home", userId: "user2"},
    {page: "/about", userId: "user1"},
    {page: "/home", userId: "user1"},  // повторний візит
    ...
]

for visit in visits do
    updateUnique(cmhll, visit.page, visit.userId)

// Запит кількості унікальних користувачів для сторінки
uniqueVisitors = queryUnique(cmhll, "/home")
```

Переваги та недоліки

Переваги

- Сублінійна пам’ять: Використовує O(1/ε × log(1/δ)) пам’яті незалежно від кількості унікальних елементів.

- Постійний час операцій: Оновлення та запити виконуються за O(log(1/δ)) часу.

- Ніколи не недооцінює: Гарантує, що оцінка завжди не менша за справжню частоту.

- Простота реалізації: Алгоритм легко реалізувати та зрозуміти.

- Паралелізація: Природно підтримує паралельну обробку та злиття.

- Детерміністичні гарантії: На відміну від семплювання, дає гарантовані межі похибки.

Недоліки

- Завжди переоцінює: Може давати завищені оцінки, особливо для рідкісних елементів.

- Залежність від розподілу: Ефективність залежить від skewness розподілу частот.

- Фіксований розмір: Розмір структури фіксується при створенні та не адаптується.

- Відсутність видалення: Не підтримує операцію зменшення лічильників.

- Похибка для всіх елементів: Навіть часті елементи мають похибку оцінки.

- Вимоги до хеш-функцій: Потребує якісних незалежних хеш-функцій.

Контрольні запитання

1. Що таке Count-Min Sketch та які основні параметри визначають його точність?

2. Поясніть, чому Count-Min Sketch завжди дає оцінку не меншу за справжню частоту.

3. Як вибір ширини та глибини впливає на точність оцінок та використання пам’яті?

4. Які властивості повинні мати хеш-функції для коректної роботи алгоритму?

5. Як працює операція злиття двох Count-Min Sketch структур?

6. Що таке conservative update та як воно покращує точність?

7. Як можна використовувати Count-Min Sketch для оцінки скалярного добутку?

8. Які існують модифікації для роботи з дуже великими частотами?

9. Як реалізувати підтримку від’ємних оновлень у Count-Min Sketch?

10. Які альтернативні sketch-структури існують та їх переваги/недоліки порівняно з Count-Min?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження Count-Min Sketch для аналізу потоків даних

2. Мета

Вивчити принципи роботи імовірнісних структур даних для підрахунку частот, реалізувати Count-Min Sketch, дослідити залежність точності від параметрів та порівняти з точними методами підрахунку.

3. Завдання

Реалізувати Count-Min Sketch з підтримкою основних операцій, створити візуалізацію розподілу колізій, провести експерименти з різними параметрами та типами даних, реалізувати розширення для часових вікон.

4. Опис

Лабораторна робота передбачає поетапну реалізацію Count-Min Sketch, включаючи генерацію незалежних хеш-функцій, базові операції update/query, візуалізацію внутрішнього стану, порівняння з точними методами. Студенти досліджують вплив параметрів на точність та пам’ять.

5. Порядок виконання

- Реалізувати базову структуру Count-Min Sketch
- Створити набір незалежних хеш-функцій
- Реалізувати операції update та query
- Додати conservative update
- Створити візуалізацію заповнення масиву
- Реалізувати операцію merge
- Додати підтримку heavy hitters
- Створити sliding window версію
- Провести експерименти з різними потоками даних
- Порівняти з HashMap за точністю та пам’яттю

6. Результат

Працююча реалізація Count-Min Sketch з візуалізацією, графіки залежності похибки від параметрів, порівняння використання пам’яті з точними методами, аналіз ефективності для різних розподілів даних, рекомендації щодо вибору параметрів.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас CountMinSketch з базовими операціями
- Генератор незалежних хеш-функцій
- Реалізація conservative update
- Модуль для знаходження heavy hitters
- Візуалізація внутрішнього стану

2. Продемонструвати роботу додатку:

- Підрахунок частот у текстовому потоці
- Виявлення найчастіших IP-адрес у логах
- Порівняння точності з різними параметрами
- Візуалізація розподілу колізій
- Робота sliding window версії

3. Бути готовим відповісти на питання:

- Як вибрати оптимальні параметри для конкретної задачі?
- Чому використовується мінімум серед оцінок?
- Як колізії впливають на точність?
- Коли Count-Min Sketch ефективніший за точні методи?

Тести

1. Яка операція використовується для отримання фінальної оцінки в Count-Min Sketch?
- 1) Максимум серед всіх лічильників
- 2) Мінімум серед всіх лічильників для елемента
- 3) Середнє арифметичне
- 4) Медіана

2. Що визначає параметр ε в Count-Min Sketch?
- 1) Ймовірність помилки
- 2) Відносну похибку оцінки
- 3) Кількість хеш-функцій
- 4) Розмір пам’яті

3. Яка складність операції update?
- 1) O(1)
- 2) O(log n)
- 3) O(d), де d - глибина
- 4) O(w), де w - ширина

4. Count-Min Sketch гарантує:
- 1) Точний підрахунок
- 2) Оцінка ⩾ справжня частота
- 3) Оцінка ⩽ справжня частота
- 4) Оцінка = справжня частота ± 1

5. Для злиття двох Count-Min Sketch потрібно:
- 1) Перерахувати всі частоти
- 2) Поелементно додати лічильники
- 3) Взяти максимум лічильників

- 4) Створити новий sketch

Відповіді:

1-2,

2-2,

3-3,

4-2,

5-2

Тести підвищеної складності

1. Чому Count-Min завжди переоцінює частоти?

- 1) Помилка в алгоритмі

- 2) Колізії хеш-функцій додають чужі лічильники

- 3) Неточність обчислень

- 4) Обмеження розрядності

2. Як conservative update покращує точність?

- 1) Використовує більше пам’яті

- 2) Оновлює тільки мінімальні лічильники

- 3) Додає випадковий шум

- 4) Збільшує кількість хеш-функцій

3. Яка пам’ять потрібна для $\varepsilon=0.001$ та $\delta=0.01$?

- 1) O(1000)

- 2) O(10000)

- 3) $O(e/\varepsilon \times \ln(1/\delta)) \approx 12000$

- 4) $O(1/\varepsilon^2)$

4. Що відбувається при дуже великій кількості унікальних елементів?

- 1) Структура автоматично розширюється

- 2) Похибка зростає через більше колізій

- 3) Алгоритм перестає працювати

- 4) Точність покращується

5. Як Count-Min-Log зменшує використання пам’яті?

- 1) Стискає дані

- 2) Зберігає логарифми замість лічильників
- 3) Використовує менше хеш-функцій
- 4) Видаляє старі дані

6. Для яких розподілів Count-Min найефективніший?
- 1) Рівномірний розподіл
- 2) Zipf розподіл (skewed)
- 3) Нормальний розподіл
- 4) Будь-який розподіл

7. Як реалізувати підтримку від'ємних значень?
- 1) Використовувати знакові лічильники
- 2) Зберігати два sketch'і - для + та -
- 3) Неможливо реалізувати
- 4) Використовувати float замість int

8. Яка основна перевага над семплюванням?
- 1) Менше пам'яті
- 2) Детерміністичні гарантії похибки
- 3) Швидші операції
- 4) Простіша реалізація

9. Що таке Count-Mean-Min Sketch?
- 1) Помилкова назва
- 2) Використовує медіану замість мінімуму
- 3) Віднімає середнє для зменшення bias
- 4) Комбінація з Mean Sketch

10. Як вибрати оптимальну кількість хеш-функцій?
- 1) Завжди використовувати 5
- 2) $\ln(1/\delta)$ для ймовірності δ
- 3) Чим більше, тим краще
- 4) Залежить від розміру даних

Відповіді:

1. 2) Кілька елементів можуть хешуватися в одну комірку, збільшуючи лічильник

2. 2) Оновлюються тільки ті лічильники, що дають мінімальну оцінку

3. 3) Формула дає приблизно 2718/0.001 × ln(100) ≈ 12500 комірок

4. 2) Більше елементів означає більше колізій та вищу похибку

5. 2) Логарифмічне представлення дозволяє зберігати великі числа в малій кількості біт

6. 2) Skewed розподіли мають менше унікальних частих елементів, що зменшує колізії

7. 2) Окремі sketch'і для додатних та від'ємних оновлень дозволяють коректно працювати

8. 2) На відміну від випадкового семплювання, дає гарантовані межі похибки

9. 3) Віднімання оціненого середнього зменшує систематичне завищення

10. 2) Кількість хеш-функцій визначається бажаною ймовірністю перевищення похибки

Завдання для самостійної роботи

1. Реалізуйте Count-Min Sketch з підтримкою від'ємних оновлень використовуючи два окремих sketch'і.

2. Створіть адаптивну версію, що динамічно змінює розмір при зростанні похибки.

3. Розробіть паралельну версію для обробки потоків на багатоядерних системах.

4. Реалізуйте Hierarchical Count-Min для ефективних range queries.

5. Створіть Count-Min-Mean Sketch для зменшення bias в оцінках.

6. Розробіть версію з компресією для зменшення використання пам'яті.

7. Реалізуйте персистентну версію з підтримкою історії змін.

8. Створіть розподілену версію для обробки даних на кластері.

9. Розробіть гібрид Count-Min та Bloom Filter для membership queries.

10. Дослідіть використання машинного навчання для оптимізації параметрів.

Теми лабораторних робіт

1. Базова реалізація Count-Min Sketch та дослідження точності.
2. Порівняльний аналіз різних sketch-структур для підрахунку частот.
3. Розробка системи моніторингу мережевого трафіку на основі Count-Min.
4. Створення real-time аналітики для потокових даних.
5. Реалізація розподіленого Count-Min для MapReduce.
6. Інтеграція Count-Min в СУБД для оптимізації запитів.
7. Розробка системи виявлення аномалій з використанням sketch'ів.
8. Створення бібліотеки sketch-структур для різних мов програмування.
9. Дослідження застосування Count-Min для біоінформатики.
10. Оптимізація Count-Min для GPU та порівняння продуктивності.

Sketch-структури: HyperLogLog

Теоретичні відомості

HyperLogLog є імовірнісною структурою даних для оцінки кардинальності (кількості унікальних елементів) у великих множинах, використовуючи надзвичайно малу кількість пам'яті. Розроблений Філіпом Флажоле, Еріком Фурманом, Олів'є Гандуе та Фредеріком Мюньє у 2007 році, цей алгоритм є вдосконаленням попереднього алгоритму LogLog і забезпечує точнішу оцінку з меншою дисперсією.

HyperLogLog базується на спостереженні за розподілом старших нулів у хеш-значеннях елементів. Інтуїтивно, якщо ми бачимо хеш-значення з великою кількістю старших нулів, це свідчить про велику кількість унікальних елементів. Наприклад, якщо найдовша послідовність старших нулів, яку ми спостерігали, дорівнює k, то ймовірно ми побачили приблизно 2^k унікальних елементів.

HyperLogLog використовує гармонічне середнє замість арифметичного, що дозволяє зменшити вплив викидів. Основна формула для оцінки кардинальності:

$$E = \frac{\alpha_m \cdot m^2}{\sum_{j=1}^{m} 2^{-M[j]}}$$

де: - m - кількість регістрів (buckets), зазвичай 2^b для деякого b - $M[j]$ - значення j-го регістра - α_m - константа корекції зміщення, що залежить від m

Константа корекції визначається як:

$$\alpha_m = \begin{cases} 0.673 & \text{якщо } m = 16 \\ 0.697 & \text{якщо } m = 32 \\ 0.709 & \text{якщо } m = 64 \\ \dfrac{0.7213}{1 + 1.079/m} & \text{якщо } m \geq 128 \end{cases}$$

Структура даних

```
struct HyperLogLog {
    m: integer              // кількість регістрів (степінь 2)
    b: integer              // кількість біт для індексу (m = 2^b)
    M: array[m] of byte     // масив регістрів
    alpha: float            // константа корекції
}
```

Ініціалізація

Процес створення та ініціалізації HyperLogLog структури:

```
function initHyperLogLog(precision: integer): HyperLogLog
    // precision визначає точність (зазвичай від 4 до 16)
    b = precision
    m = 2^b

    // Ініціалізуємо всі регістри нулями
    M = array[m]
    for i = 0 to m - 1 do
        M[i] = 0

    // Обчислюємо константу корекції
    if m == 16 then
        alpha = 0.673
    else if m == 32 then
        alpha = 0.697
    else if m == 64 then
        alpha = 0.709
    else
        alpha = 0.7213 / (1 + 1.079 / m)

    hll = HyperLogLog(
        m = m,
        b = b,
```

```
        M = M,
        alpha = alpha
    )

    return hll
```

Операція ADD (додавання елемента)

Основна операція додавання елемента до HyperLogLog:

```
function add(hll: HyperLogLog, element: any)
    // Обчислюємо хеш елемента (наприклад, 64-бітний)
    hash = hash64(element)

    // Використовуємо перші b біт для вибору регістра
    j = hash >> (64 - hll.b)  // індекс регістра (від 0 до m-1)

    // Решта біт використовується для підрахунку старших нулів
    w = hash << hll.b  // видаляємо використані біти

    // Підраховуємо кількість старших нулів + 1
    rho = leadingZeroCount(w) + 1

    // Оновлюємо регістр, якщо нове значення більше
    if rho > hll.M[j] then
        hll.M[j] = rho

function leadingZeroCount(value: uint64): integer
    // Підраховуємо кількість старших нулів
    if value == 0 then
        return 64

    count = 0
    mask = 1 << 63  // старший біт

    while (value & mask) == 0 do
        count += 1
        mask = mask >> 1

    return count
```

Операція CARDINALITY (оцінка кардинальності)

Обчислення оцінки кількості унікальних елементів:

```
function cardinality(hll: HyperLogLog): float
    // Обчислюємо сире значення
    rawEstimate = computeRawEstimate(hll)
```

```
    // Застосовуємо корекції для малих та великих значень
    if rawEstimate <= 2.5 * hll.m then
        // Корекція для малих значень
        zeros = countZeros(hll.M)
        if zeros != 0 then
            return hll.m * ln(hll.m / zeros)

    if rawEstimate <= (1.0/30.0) * (1 << 32) then
        // Немає корекції
        return rawEstimate
    else
        // Корекція для великих значень
        return -1 * (1 << 32) * ln(1 - rawEstimate / (1 << 32))

function computeRawEstimate(hll: HyperLogLog): float
    // Обчислюємо суму 2^(-M[j])
    sum = 0.0
    for j = 0 to hll.m - 1 do
        sum += 1.0 / (1 << hll.M[j])

    // Застосовуємо формулу HyperLogLog
    estimate = hll.alpha * hll.m * hll.m / sum
    return estimate

function countZeros(M: array of byte): integer
    count = 0
    for value in M do
        if value == 0 then
            count += 1
    return count
```

Операція MERGE (об’єднання)

HyperLogLog підтримує ефективне об’єднання кількох структур:

```
function merge(hll1: HyperLogLog, hll2: HyperLogLog): HyperLogLog
    // Перевіряємо сумісність
    if hll1.m != hll2.m then
        throw Error("Incompatible HyperLogLog structures")

    // Створюємо новий HyperLogLog
    result = initHyperLogLog(hll1.b)

    // Об'єднуємо регістри, беручи максимум
    for j = 0 to result.m - 1 do
        result.M[j] = max(hll1.M[j], hll2.M[j])
```

```
    return result
```

HyperLogLog++ покращення

1. Google розробив покращену версію HyperLogLog++, яка включає кілька оптимізацій:

Sparse представлення

Для малої кількості елементів використовується розріджене представлення:

```
struct SparseHyperLogLog {
    tmpSet: set of (index, rho) pairs
    sparseListSize: integer
    maxSparseSize: integer
}

function addSparse(shll: SparseHyperLogLog, element: any)
    hash = hash64(element)
    index = hash >> (64 - shll.b)
    w = hash << shll.b
    rho = leadingZeroCount(w) + 1

    // Додаємо або оновлюємо пару
    existingPair = shll.tmpSet.find(index)
    if existingPair != null then
        if rho > existingPair.rho then
            existingPair.rho = rho
    else
        shll.tmpSet.add((index, rho))
        shll.sparseListSize += 1

    // Перевіряємо чи потрібно перейти до щільного представлення
    if shll.sparseListSize > shll.maxSparseSize then
        return convertToDense(shll)
```

64-бітне хешування

Використання 64-бітних хеш-функцій для кращої точності:

```
function hash64(element: any): uint64
    // Використовуємо MurmurHash3 або подібну функцію
    return murmurHash3_64bit(element)
```

Bias корекція

Емпірична корекція для зменшення систематичної похибки:

```
function biasCorrection(rawEstimate: float, m: integer): float
    // Використовуємо попередньо обчислені таблиці корекції
    if rawEstimate <= 5 * m then
        bias = getBiasEstimate(rawEstimate, m)
        return rawEstimate - bias

    return rawEstimate

function getBiasEstimate(estimate: float, m: integer): float
    // Інтерполяція з таблиці емпіричних значень
    biasData = loadBiasData(m)
    return interpolate(biasData, estimate)
```

Оптимізації реалізації

1. Використання lookup таблиць

Для швидкого підрахунку старших нулів:

```
// Попередньо обчислена таблиця для 8-бітних значень
leadingZeroTable = [
    8, 7, 6, 6, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    // ... продовження для всіх 256 значень
]

function fastLeadingZeroCount(value: uint64): integer
    if value == 0 then
        return 64

    // Перевіряємо байти починаючи зі старшого
    for i = 7 downto 0 do
        byte = (value >> (i * 8)) & 0xFF
        if byte != 0 then
            return (7 - i) * 8 + leadingZeroTable[byte]

    return 64
```

2. Паралельна обробка

Розподілення роботи між потоками:

```
function parallelAdd(hll: HyperLogLog, elements: array of any, numThr
eads: integer)
    // Створюємо локальні HLL для кожного потоку
    localHLLs = array[numThreads]
    for i = 0 to numThreads - 1 do
        localHLLs[i] = initHyperLogLog(hll.b)
```

```
    // Розподіляємо елементи між потоками
    chunkSize = ceil(elements.length / numThreads)

    parallel for t = 0 to numThreads - 1 do
        start = t * chunkSize
        end = min((t + 1) * chunkSize, elements.length)

        for i = start to end - 1 do
            add(localHLLs[t], elements[i])

    // Об'єднуємо результати
    for i = 0 to numThreads - 1 do
        hll = merge(hll, localHLLs[i])
```

3. Стиснення даних

Оскільки значення регістрів зазвичай малі, можна використовувати стиснення:

```
struct CompressedHyperLogLog {
    m: integer
    b: integer
    compressedData: byte array
    compressionScheme: string
}

function compress(hll: HyperLogLog): CompressedHyperLogLog
    // Використовуємо variable-length encoding
    compressed = []

    for value in hll.M do
        if value <= 15 then
            // 4 біти для малих значень
            compressed.appendBits(value, 4)
        else
            // Escape sequence + повне значення
            compressed.appendBits(15, 4)
            compressed.appendBits(value - 15, 6)

    return CompressedHyperLogLog(
        m = hll.m,
        b = hll.b,
        compressedData = compressed.toByteArray(),
        compressionScheme = "variable-length"
    )
```

Область застосування

HyperLogLog знаходить широке застосування в системах, де потрібно оцінювати кардинальність великих множин:

- Аналітика веб-трафіку: Підрахунок унікальних відвідувачів сайту без збереження всіх IP-адрес.

- Бази даних: Оцінка кардинальності для оптимізації планів виконання запитів.

- Соціальні мережі: Підрахунок унікальних користувачів, які взаємодіяли з контентом.

- Моніторинг систем: Відстеження кількості унікальних подій або помилок.

- Розподілені системи: Агрегація унікальних елементів з багатьох вузлів.

- Потокова обробка: Оцінка кардинальності в реальному часі для потоків даних.

Приклади застосування

Приклад 1: Базова робота HyperLogLog

Розглянемо роботу HyperLogLog з m=16 регістрами (b=4 біти):

```
Елементи для додавання: "user1", "user2", "user3", "user1" (повтор)

Ініціалізація:
M = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Додавання "user1":
hash("user1") = 0b1101001110101001...  (64 біти)
j = 0b1101 = 13  (перші 4 біти)
w = 0b001110101001...  (решта 60 біт)
ρ = 3  (дві старші нулі + 1)
M[13] = 3

Додавання "user2":
hash("user2") = 0b0010110011001010...
j = 0b0010 = 2
w = 0b110011001010...
ρ = 1  (нуль старших нулів + 1)
M[2] = 1

Додавання "user3":
hash("user3") = 0b1000000010111001...
j = 0b1000 = 8
w = 0b000010111001...
ρ = 5  (чотири старші нулі + 1)
```

```
M[8] = 5

Додавання "user1" (повтор):
hash("user1") = 0b1101001110101001...
j = 13, ρ = 3
M[13] вже дорівнює 3, не змінюється

Фінальний стан:
M = [0, 0, 1, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 3, 0, 0]

Оцінка кардинальності:
sum = 2^0 + 2^0 + 2^(-1) + 2^0 + ... + 2^(-5) + ... + 2^(-3) + ...
    = 1 + 1 + 0.5 + 1 + ... + 0.03125 + ... + 0.125 + ...
    ≈ 14.65625

E = 0.673 × 16² / 14.65625 ≈ 2.95

Результат: ≈ 3 унікальні елементи
```

Приклад 2: Підрахунок унікальних IP-адрес

Система моніторингу веб-сервера:

```
struct WebAnalytics {
    dailyVisitors: HyperLogLog
    hourlyVisitors: array[24] of HyperLogLog
    pathVisitors: HashMap<string, HyperLogLog>
}

function processLogEntry(analytics: WebAnalytics, entry: LogEntry)
    ipAddress = entry.ipAddress
    hour = entry.timestamp.getHour()
    path = entry.requestPath

    // Додаємо до щоденної статистики
    analytics.dailyVisitors.add(ipAddress)

    // Додаємо до погодинної статистики
    analytics.hourlyVisitors[hour].add(ipAddress)

    // Додаємо до статистики шляхів
    if path not in analytics.pathVisitors then
        analytics.pathVisitors[path] = initHyperLogLog(14)
    analytics.pathVisitors[path].add(ipAddress)

// Отримання статистики
function getDailyStats(analytics: WebAnalytics): Stats
```

```
    return Stats(
        uniqueVisitors = analytics.dailyVisitors.cardinality(),
        peakHour = findPeakHour(analytics.hourlyVisitors),
        popularPaths = getTopPaths(analytics.pathVisitors, 10)
    )

// Приклад використання
analytics = WebAnalytics(
    dailyVisitors = initHyperLogLog(14),  // похибка ~0.8%
    hourlyVisitors = [initHyperLogLog(12) for i in 0..23],
    pathVisitors = HashMap()
)

// Обробка логів
logs = [
    {ip: "192.168.1.1", timestamp: "2024-01-01 10:00", path: "/home"}
,
    {ip: "192.168.1.2", timestamp: "2024-01-01 10:05", path: "/produc
ts"},
    {ip: "192.168.1.1", timestamp: "2024-01-01 10:10", path: "/produc
ts"},
    ...
]

for entry in logs do
    processLogEntry(analytics, entry)

stats = getDailyStats(analytics)
// Результат: 150,000 унікальних відвідувачів (використовуючи лише 16
KB пам'яті)
```

Приклад 3: Розподілена система

Агрегація даних з кількох серверів:

```
struct DistributedCounter {
    nodeId: string
    localHLL: HyperLogLog
    mergeInterval: Duration
}

function collectMetrics(counter: DistributedCounter, centralServer: S
erver)
    while true do
        // Збираємо локальні метрики
        events = collectLocalEvents(counter.mergeInterval)
```

```
        for event in events do
            counter.localHLL.add(event.userId)

        // Відправляємо на центральний сервер
        serialized = serialize(counter.localHLL)
        centralServer.send(counter.nodeId, serialized)

        // Очищуємо локальний HLL для наступного інтервалу
        counter.localHLL = initHyperLogLog(14)

        sleep(counter.mergeInterval)

// Центральний сервер
function aggregateMetrics(server: Server)
    globalHLL = initHyperLogLog(14)
    nodeHLLs = HashMap<string, HyperLogLog>()

    while true do
        message = server.receive()
        nodeId = message.nodeId
        nodeHLL = deserialize(message.data)

        // Зберігаємо або оновлюємо HLL вузла
        if nodeId in nodeHLLs then
            nodeHLLs[nodeId] = merge(nodeHLLs[nodeId], nodeHLL)
        else
            nodeHLLs[nodeId] = nodeHLL

        // Переобчислюємо глобальну метрику
        globalHLL = initHyperLogLog(14)
        for (_, hll) in nodeHLLs do
            globalHLL = merge(globalHLL, hll)

        totalUniqueUsers = globalHLL.cardinality()
        updateDashboard(totalUniqueUsers)
```

Приклад 4: A/B тестування

Використання HyperLogLog для аналізу експериментів:

```
struct ABTestAnalyzer {
    controlGroup: HyperLogLog
    testGroup: HyperLogLog
    overlapHLL: HyperLogLog  // користувачі в обох групах
}

function analyzeABTest(analyzer: ABTestAnalyzer): ABTestResults
```

```
    controlSize = analyzer.controlGroup.cardinality()
    testSize = analyzer.testGroup.cardinality()

    // Обчислюємо об'єднання для загальної кількості
    union = merge(analyzer.controlGroup, analyzer.testGroup)
    totalUsers = union.cardinality()

    // Оцінка перетину (користувачі в обох групах)
    // |A ∩ B| = |A| + |B| - |A ∪ B|
    overlap = controlSize + testSize - totalUsers

    return ABTestResults(
        controlUsers = controlSize,
        testUsers = testSize,
        totalUsers = totalUsers,
        overlapUsers = overlap,
        overlapPercentage = overlap / totalUsers * 100
    )

// Використання
analyzer = ABTestAnalyzer(
    controlGroup = initHyperLogLog(14),
    testGroup = initHyperLogLog(14),
    overlapHLL = initHyperLogLog(14)
)

// Обробка подій
events = [
    {userId: "u1", group: "control", action: "click"},
    {userId: "u2", group: "test", action: "purchase"},
    {userId: "u1", group: "test", action: "view"},  // користувач в о
бох групах
    ...
]

for event in events do
    if event.group == "control" then
        analyzer.controlGroup.add(event.userId)
    else
        analyzer.testGroup.add(event.userId)

results = analyzeABTest(analyzer)
// Результат: 50K в контрольній, 48K в тестовій, 3K перетин
```

Переваги та недоліки

Переваги

- Надзвичайно мала пам’ять: Використовує лише O(m log log n) біт, де m - кількість регістрів, незалежно від кількості елементів.

- Постійний час операцій: Додавання елемента та оцінка кардинальності виконуються за O(1).

- Висока точність: Стандартна похибка приблизно 1.04/√m, тобто для m=2^14 похибка ~0.81%.

- Підтримка об’єднання: Можливість об’єднувати HLL структури для агрегації даних.

- Детерміністичність: Для одних і тих же даних завжди дає однаковий результат.

- Простота реалізації: Алгоритм відносно простий у реалізації та розумінні.

Недоліки

- Лише оцінка: Не дає точного значення, а лише наближення з певною похибкою.

- Не підтримує видалення: Неможливо видалити елемент після додавання.

- Фіксований розмір: Точність визначається при ініціалізації та не може бути змінена.

- Не зберігає елементи: Неможливо отримати список унікальних елементів.

- Похибка для малих множин: Менш ефективний для дуже малої кількості елементів.

- Залежність від хеш-функції: Якість оцінки залежить від якості хеш-функції.

Контрольні запитання

1. Яка основна ідея алгоритму HyperLogLog для оцінки кардинальності?

2. Чому HyperLogLog використовує гармонічне середнє замість арифметичного?

3. Як вибір кількості регістрів m впливає на точність та використання пам’яті?

4. Що таке константа корекції α та як вона обчислюється?

5. Поясніть процес додавання елемента до HyperLogLog структури.

6. Які корекції застосовуються для малих та великих значень кардинальності?

7. Як працює операція об’єднання двох HyperLogLog структур?

8. Які основні покращення включає HyperLogLog++ порівняно з оригінальним алгоритмом?

9. Чому для малих множин використовується інша формула оцінки?

10. Як можна оптимізувати HyperLogLog для роботи з потоками даних?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження HyperLogLog для оцінки кардинальності

2. Мета

Вивчити принципи імовірнісної оцінки кардинальності, реалізувати алгоритм HyperLogLog, дослідити залежність точності від параметрів, порівняти з точними методами підрахунку унікальних елементів.

3. Завдання

Реалізувати HyperLogLog з різними значеннями точності, створити візуалізацію розподілу значень у регістрах, провести експерименти з різними типами даних, реалізувати оптимізації HyperLogLog++.

4. Опис

Лабораторна робота включає реалізацію базового алгоритму HyperLogLog, функцій для підрахунку старших нулів, корекцій для граничних випадків, операції об’єднання. Студенти досліджують точність оцінок та оптимізують реалізацію.

5. Порядок виконання

- Реалізувати структуру HyperLogLog з конфігурованою точністю
- Створити ефективну функцію підрахунку старших нулів
- Реалізувати операції add та cardinality з корекціями
- Додати операцію merge для об’єднання структур
- Реалізувати sparse режим для малих множин
- Створити візуалізацію заповнення регістрів
- Провести тестування на різних датасетах
- Порівняти з HashSet за точністю та пам’яттю

- Реалізувати серіалізацію/десеріалізацію

- Створити бенчмарки продуктивності

6. Результат

Працююча реалізація HyperLogLog з різними режимами точності, графіки залежності похибки від кількості елементів, порівняння використання пам’яті з точними методами, візуалізація внутрішнього стану, рекомендації щодо вибору параметрів для різних застосувань.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас HyperLogLog з базовими операціями

- Оптимізована функція leadingZeroCount

- Реалізація bias корекції

- Sparse та dense режими

- Модуль серіалізації

2. Продемонструвати роботу додатку:

- Оцінка кардинальності для різних датасетів

- Візуалізація розподілу значень у регістрах

- Порівняння точності при різних m

- Робота операції merge

- Аналіз використання пам’яті

3. Бути готовим відповісти на питання:

- Чому використовується позиція першої одиниці?

- Як вибрати оптимальне значення m?

- Коли sparse режим ефективніший?

- Як якість хеш-функції впливає на результат?

Тести

1. Яка стандартна похибка HyperLogLog з m регістрами?

- 1) $1/\sqrt{m}$

- 2) $1.04/\sqrt{m}$

- 3) $2/m$

- 4) log(m)/m

2. Що зберігається в кожному регістрі HyperLogLog?

- 1) Кількість елементів
- 2) Хеш-значення
- 3) Максимальна кількість старших нулів + 1
- 4) Унікальні елементи

3. Скільки біт потрібно для регістра при 64-бітному хеші?

- 1) 64 біти
- 2) 32 біти
- 3) 6-8 біт зазвичай достатньо
- 4) 1 біт

4. Як об’єднуються два HyperLogLog?

- 1) Додаються значення регістрів
- 2) Береться максимум для кожного регістра
- 3) Береться мінімум для кожного регістра
- 4) Обчислюється середнє

5. Для якої кількості елементів застосовується LinearCounting корекція?

- 1) Великої кількості
- 2) Малої кількості (⩽ 2.5m)
- 3) Завжди
- 4) Ніколи

Відповіді:

1-2,

2-3,

3-3,

4-2,

5-2

Тести підвищеної складності

1. Чому гармонічне середнє краще арифметичного для HyperLogLog?

- 1) Швидше обчислюється

- 2) Менш чутливе до великих викидів
- 3) Дає більшу оцінку
- 4) Простіше реалізувати

2. Яка ймовірність побачити k старших нулів?
- 1) 1/k
- 2) 1/2^k
- 3) 1/2^(k+1)
- 4) 2^k

3. Що таке bias в контексті HyperLogLog?
- 1) Помилка округлення
- 2) Систематична похибка оцінки
- 3) Випадкова похибка
- 4) Похибка хешування

4. Коли використовується large range корекція?
- 1) Для малих значень
- 2) Коли оцінка близька до 2^32
- 3) Завжди
- 4) Для від’ємних значень

5. Яка складність пам’яті HyperLogLog?
- 1) O(n)
- 2) O(log n)
- 3) O(1)
- 4) O(m log log n)

6. Що покращує sparse режим у HyperLogLog++?
- 1) Точність для великих множин
- 2) Ефективність для малих множин
- 3) Швидкість об’єднання
- 4) Стиснення даних

7. Як вибирається оптимальне значення m?
- 1) Завжди 2^14

- 2) Залежно від бажаної точності та пам’яті

- 3) Залежно від кількості елементів

- 4) Випадково

8. Що означає precision=14 в HyperLogLog?

- 1) 14% похибка

- 2) 14 біт для хешу

- 3) 2^14 регістрів

- 4) 14 байт пам’яті

9. Чому HyperLogLog не підтримує видалення?

- 1) Технічна складність

- 2) Зберігається лише максимум старших нулів

- 3) Обмеження хеш-функції

- 4) Для економії пам’яті

10. Яка перевага 64-бітного хешування над 32-бітним?

- 1) Швидше працює

- 2) Більший діапазон без колізій

- 3) Менше пам’яті

- 4) Простіша реалізація

Відповіді:

1. 2) Гармонічне середнє надає меншу вагу великим значенням, зменшуючи вплив викидів

2. 3) Ймовірність отримати рівно k старших нулів дорівнює 1/2^(k+1)

3. 2) Систематичне відхилення оцінки від справжнього значення

4. 2) Корекція застосовується коли оцінка наближається до межі 32-бітного простору

5. 4) m регістрів по log log n біт кожен для підрахунку старших нулів

6. 2) Зберігає лише ненульові регістри, ефективніше для малої кількості унікальних елементів

7. 2) Баланс між бажаною точністю (1.04/√m) та використанням пам’яті

8. 3) Використовується 2^14 = 16384 регістри

9. 2) Неможливо визначити чи максимум прийшов від видаленого елемента

10. 2) Зменшує ймовірність хеш-колізій для великих датасетів

Завдання для самостійної роботи

1. Реалізуйте sliding window HyperLogLog для підрахунку унікальних елементів за останній час.

2. Створіть розподілену версію HyperLogLog для роботи на кластері.

3. Розробіть адаптивну версію, що автоматично вибирає precision на основі даних.

4. Реалізуйте HyperLogLog з підтримкою зважених елементів.

5. Створіть версію з шифруванням для конфіденційних даних.

6. Розробіть механізм стиснення для довготривалого зберігання HLL.

7. Реалізуйте continuous HyperLogLog для потокової обробки.

8. Створіть візуальний дебагер для аналізу роботи алгоритму.

9. Розробіть HyperLogLog-based систему дедуплікації.

10. Дослідіть можливість використання машинного навчання для покращення bias корекції.

Теми лабораторних робіт

1. Базова реалізація HyperLogLog та аналіз точності.

2. Порівняння HyperLogLog, Linear Counting та LogLog алгоритмів.

3. Розробка системи унікальних метрик для веб-аналітики.

4. Створення розподіленого лічильника унікальних подій.

5. Реалізація HyperLogLog++ з усіма оптимізаціями Google.

6. Інтеграція HyperLogLog в систему моніторингу.

7. Розробка бібліотеки sketch-алгоритмів для різних мов.

8. Дослідження застосування HyperLogLog для геномних даних.

9. Створення системи A/B тестування на основі HyperLogLog.

10. Оптимізація HyperLogLog для вбудованих систем.

14.2. Алгоритми для потоків

Уявіть, що дані — це не статичний набір файлів на диску, а нескінченна ріка інформації, що протікає через вашу систему: потік кліків на вебсайті, потік

транзакцій або вимірювань із сенсорів. Зберігати кожен елемент такого потоку неможливо. Як у таких умовах відповісти на, здавалося б, просте запитання: “Які елементи зустрічаються найчастіше?”

Саме цю задачу, відому як пошук “важковаговиків” (heavy hitters), вирішують алгоритми для потоків. Вони працюють в умовах жорстких обмежень, обробляючи кожен елемент лише один раз і використовуючи мінімальний, фіксований обсяг пам’яті.

У цьому розділі ми розглянемо два класичні та ефективні підходи:

- **Lossy Counting:** Алгоритм, який періодично “забуває” про елементи з низькою частотою, що гарантує виявлення всіх справді частих елементів.

- **Space-Saving:** Більш точний підхід, який підтримує лічильники для обмеженої кількості елементів і динамічно замінює найменш значущі з них новими.

Ці методи дозволяють виявляти головні тренди та аномалії в реальному часі, не чекаючи на збереження та пакетну обробку даних.

Lossy Counting

Теоретичні відомості

Lossy Counting є детермінованим алгоритмом для знаходження частих елементів у потоках даних з використанням обмеженої пам’яті. Розроблений Гуптою Мотвані та Сімоне Манку у 2002 році, цей алгоритм гарантує, що всі справжні часті елементи будуть знайдені, при цьому може включати деякі помилкові елементи, частота яких трохи нижча за поріг.

Lossy Counting працює з потоком даних, розділяючи його на логічні вікна фіксованого розміру. Алгоритм підтримує структуру даних, що містить елементи та їх оцінені частоти, періодично видаляючи рідкісні елементи для економії пам’яті.

Цей алгоритм працює так:

1. Розділення потоку: Увесь потік даних ділиться на уявні блоки (або “вікна”).

2. Підрахунок у пам’яті: Алгоритм зберігає список пар: `(елемент, його частота)`.

3. Обробка елементів:

- Коли надходить новий елемент, алгоритм перевіряє, чи є він у списку.

- Якщо так, його лічильник частоти збільшується на 1.

- Якщо ні, цей новий елемент додається до списку з частотою 1.

4. “Чистка” в кінці блоку: Коли поточний блок даних закінчується:

- Алгоритм проходиться по всьому списку і зменшує кожен лічильник на 1.

- Якщо лічильник будь-якого елемента стає рівним 0, цей елемент видаляється зі списку.

Назва алгоритму походить від кроку “чистки”, під час якого алгоритм “забуває” про рідкісні або застарілі елементи, чим і економить пам’ять. При цьому він гарантовано не пропустить жодного по-справжньому частого елемента, але може помилково зберегти деякі не дуже часті.

Ключові параметри алгоритму:

- s - поріг підтримки (support threshold), частка від загальної кількості елементів

- ϵ - параметр похибки, де $\epsilon < s$

- N - поточна кількість оброблених елементів - $w = \lceil 1/\epsilon \rceil$ - розмір вікна

Структура даних

```
struct LossyCounting {
    epsilon: float          // параметр похибки
    support: float          // поріг підтримки
    windowSize: integer     // w = ceil(1/epsilon)
    currentBucket: integer  // поточний номер bucket
    N: integer             // кількість оброблених елементів
    D: HashMap<item, (frequency, delta)>  // структура даних
}

struct ItemEntry {
    frequency: integer  // оцінена частота елемента
    delta: integer     // максимальна можлива похибка
}
```

Ініціалізація алгоритму

Процес створення та ініціалізації структури Lossy Counting:

```
function initLossyCounting(support: float, epsilon: float): LossyCoun
ting
    // Перевіряємо коректність параметрів
    if epsilon >= support then
        throw Error("Epsilon must be less than support")
```

```
    if epsilon <= 0 or support <= 0 or support >= 1 then
        throw Error("Invalid parameters")

    // Обчислюємо розмір вікна
    windowSize = ceil(1.0 / epsilon)

    lc = LossyCounting(
        epsilon = epsilon,
        support = support,
        windowSize = windowSize,
        currentBucket = 1,
        N = 0,
        D = new HashMap()
    )

    return lc
```

Операція PROCESS (обробка елемента)

Основна операція обробки нового елемента з потоку:

```
function processElement(lc: LossyCounting, item: any)
    // Збільшуємо лічильник елементів
    lc.N += 1

    // Визначаємо поточний bucket
    bucketId = ceil(lc.N / lc.windowSize)

    // Оновлюємо або додаємо елемент
    if item in lc.D then
        // Елемент вже існує, збільшуємо частоту
        lc.D[item].frequency += 1
    else
        // Новий елемент, додаємо з delta = bucketId - 1
        lc.D[item] = ItemEntry(
            frequency = 1,
            delta = bucketId - 1
        )

    // Перевіряємо чи закінчилось вікно
    if lc.N mod lc.windowSize == 0 then
        // Виконуємо очищення в кінці вікна
        cleanupWindow(lc, bucketId)
        lc.currentBucket = bucketId + 1
```

Операція CLEANUP (очищення)

Видалення рідкісних елементів в кінці кожного вікна:

```
function cleanupWindow(lc: LossyCounting, currentBucket: integer)
    // Створюємо список елементів для видалення
    toDelete = []

    // Перевіряємо кожен елемент
    for (item, entry) in lc.D do
        // Видаляємо елементи з f + △ ≤ b_current
        if entry.frequency + entry.delta <= currentBucket then
            toDelete.append(item)

    // Видаляємо елементи
    for item in toDelete do
        lc.D.remove(item)
```

Операція QUERY (запит частих елементів)

Отримання всіх елементів з частотою вище порогу:

```
function getFrequentItems(lc: LossyCounting, minSupport: float = null
): list of (item, frequency)
    // Використовуємо заданий поріг або поріг з ініціалізації
    if minSupport == null then
        minSupport = lc.support

    // Обчислюємо абсолютний поріг
    threshold = (minSupport - lc.epsilon) * lc.N

    results = []

    // Перевіряємо всі елементи
    for (item, entry) in lc.D do
        // Повертаємо елементи з частотою >= threshold
        if entry.frequency >= threshold then
            results.append((item, entry.frequency))

    // Сортуємо за частотою
    results.sortByDescending(pair => pair.frequency)

    return results
```

Lossy Counting надає наступні гарантії:

1. **Повнота**: Всі елементи з справжньою частотою $f_i \geq sN$ будуть у результаті

2. **Точність**: Жоден елемент з частотою $f_i < (s - \epsilon)N$ не буде у результаті

3. **Похибка**: Оцінена частота відрізняється від справжньої не більше ніж на ϵN

Оцінка використання пам'яті

Теоретична верхня межа кількості елементів у структурі D:

$$|D| \leq \frac{1}{\epsilon} \log(\epsilon N)$$

На практиці використання пам'яті зазвичай набагато менше через видалення рідкісних елементів.

Модифікації та оптимізації

1. Lossy Counting з підрахунком top-k

Модифікація для знаходження k найчастіших елементів:

```
function processElementTopK(lc: LossyCounting, item: any, k: integer)
    // Звичайна обробка
    processElement(lc, item)

    // Додаткове очищення для підтримки розміру
    if lc.D.size() > k / lc.epsilon then
        // Знаходимо k-й елемент за частотою
        frequencies = []
        for (item, entry) in lc.D do
            frequencies.append(entry.frequency + entry.delta)

        frequencies.sort()
        kthFrequency = frequencies[k] if k < frequencies.length else
0

        // Видаляємо елементи нижче k-го
        toDelete = []
        for (item, entry) in lc.D do
            if entry.frequency + entry.delta < kthFrequency then
                toDelete.append(item)

        for item in toDelete do
            lc.D.remove(item)
```

2. Ієрархічний Lossy Counting

Для обробки ієрархічних даних (наприклад, IP-адреси з підмережами):

```
struct HierarchicalLossyCounting {
    levels: array of LossyCounting  // окремий LC для кожного рівня
    hierarchyFunction: function     // функція отримання рівнів
}
```

```
function processHierarchical(hlc: HierarchicalLossyCounting, item: an
y)
    // Отримуємо всі рівні ієрархії
    levels = hlc.hierarchyFunction(item)

    // Обробляємо на кожному рівні
    for i = 0 to levels.length - 1 do
        processElement(hlc.levels[i], levels[i])

// Приклад для IP-адрес
function ipHierarchy(ip: string): array of string
    parts = ip.split(".")
    levels = []

    // Рівень 1: перший октет (клас A)
    levels.append(parts[0] + ".*.*.*")

    // Рівень 2: два октети (клас B)
    levels.append(parts[0] + "." + parts[1] + ".*.*")

    // Рівень 3: три октети (клас C)
    levels.append(parts[0] + "." + parts[1] + "." + parts[2] + ".*")

    // Рівень 4: повна адреса
    levels.append(ip)

    return levels
```

3. Паралельна версія

Розподілення обробки між кількома потоками:

```
struct ParallelLossyCounting {
    numThreads: integer
    localCounters: array of LossyCounting
    mergeLock: Mutex
}

function processParallel(plc: ParallelLossyCounting, items: stream)
    // Розподіляємо елементи між потоками
    threads = []

    for t = 0 to plc.numThreads - 1 do
        threads.append(createThread(() => {
            localLC = plc.localCounters[t]
```

```
            while item = items.getNext() do
                processElement(localLC, item)

                // Періодично об'єднуємо результати
                if localLC.N mod (localLC.windowSize * 10) == 0 then
                    plc.mergeLock.lock()
                    mergeIntoGlobal(plc, localLC)
                    plc.mergeLock.unlock()
        }))

    // Чекаємо завершення всіх потоків
    for thread in threads do
        thread.join()

    // Фінальне об'єднання
    finalMerge(plc)
```

4. Адаптивний Lossy Counting

Версія, що автоматично налаштовує параметри:

```
struct AdaptiveLossyCounting {
    targetMemory: integer      // цільове використання пам'яті
    minEpsilon: float         // мінімальна похибка
    maxEpsilon: float         // максимальна похибка
    currentLC: LossyCounting  // поточна структура
}

function processAdaptive(alc: AdaptiveLossyCounting, item: any)
    // Обробляємо елемент
    processElement(alc.currentLC, item)

    // Перевіряємо використання пам'яті
    currentMemory = alc.currentLC.D.size() * ENTRY_SIZE

    if currentMemory > alc.targetMemory then
        // Збільшуємо epsilon для зменшення пам'яті
        newEpsilon = min(alc.currentLC.epsilon * 1.5, alc.maxEpsilon)
        rebuildWithNewEpsilon(alc, newEpsilon)

    else if currentMemory < alc.targetMemory * 0.5 then
        // Зменшуємо epsilon для кращої точності
        newEpsilon = max(alc.currentLC.epsilon * 0.75, alc.minEpsilon
)
        rebuildWithNewEpsilon(alc, newEpsilon)
```

Область застосування

Lossy Counting широко використовується для аналізу потоків даних у реальному часі:

- Мережевий моніторинг: Виявлення найактивніших IP-адрес, популярних URL, аналіз трафіку за протоколами.

- Веб-аналітика: Знаходження популярних сторінок, частих пошукових запитів, трендових тем.

- Системи рекомендацій: Відстеження популярних товарів, частих покупок, трендових продуктів.

- Аналіз логів: Виявлення частих помилок, популярних API-ендпоінтів, проблемних користувачів.

- Соціальні мережі: Знаходження трендових хештегів, популярних постів, активних користувачів.

- Фінансові системи: Виявлення частих транзакцій, підозрілих операцій, популярних інструментів.

Приклади застосування

Приклад 1: Базова робота алгоритму

Розглянемо потік символів з параметрами s=0.3, ε=0.1:

```
Потік: A, B, A, C, A, B, D, A, B, C, A, B

Параметри:
- support = 0.3 (30%)
- epsilon = 0.1 (10%)
- windowSize = ceil(1/0.1) = 10

Обробка:
Елементи 1-10 (перше вікно):
1. A: D[A] = (1, 0)
2. B: D[B] = (1, 0)
3. A: D[A] = (2, 0)
4. C: D[C] = (1, 0)
5. A: D[A] = (3, 0)
6. B: D[B] = (2, 0)
7. D: D[D] = (1, 0)
8. A: D[A] = (4, 0)
9. B: D[B] = (3, 0)
10. C: D[C] = (2, 0)

Кінець вікна 1 (bucket = 1):
```

```
Очищення: видаляємо елементи з f + Δ ≤ 1
D[D] = (1, 0), 1 + 0 ≤ 1, видаляємо D

Стан після вікна 1:
D = {A: (4, 0), B: (3, 0), C: (2, 0)}

Елементи 11-12:
11. A: D[A] = (5, 0)
12. B: D[B] = (4, 0)

Фінальний стан (N=12):
D = {A: (5, 0), B: (4, 0), C: (2, 0)}

Запит частих елементів (s=0.3):
threshold = (0.3 - 0.1) × 12 = 2.4
Результат: A(5), B(4) - обидва >= 2.4
C не повертається, бо 2 < 2.4
```

Приклад 2: Аналіз веб-логів

Система аналізу популярних сторінок сайту:

```
struct WebLogAnalyzer {
    pageCounter: LossyCounting
    userAgentCounter: LossyCounting
    referrerCounter: LossyCounting
    hourlyStats: array[24] of LossyCounting
}

function analyzeWebLog(analyzer: WebLogAnalyzer, logEntry: LogEntry)
    // Обробляємо URL сторінки
    processElement(analyzer.pageCounter, logEntry.url)

    // Обробляємо User-Agent
    processElement(analyzer.userAgentCounter, logEntry.userAgent)

    // Обробляємо referrer
    if logEntry.referrer != null then
        processElement(analyzer.referrerCounter, logEntry.referrer)

    // Обробляємо за годинами
    hour = logEntry.timestamp.getHour()
    processElement(analyzer.hourlyStats[hour], logEntry.url)

function getWebStats(analyzer: WebLogAnalyzer): WebStatistics
    return WebStatistics(
        topPages = getFrequentItems(analyzer.pageCounter, 0.01),
```

```
        topBrowsers = extractBrowsers(analyzer.userAgentCounter),
        topReferrers = getFrequentItems(analyzer.referrerCounter, 0.0
05),
        peakHours = findPeakHours(analyzer.hourlyStats)
    )

// Приклад використання
analyzer = WebLogAnalyzer(
    pageCounter = initLossyCounting(0.001, 0.0001),  // топ 0.1% стор
інок
    userAgentCounter = initLossyCounting(0.01, 0.001),
    referrerCounter = initLossyCounting(0.005, 0.0005),
    hourlyStats = [initLossyCounting(0.01, 0.001) for i in 0..23]
)

// Обробка логів
logs = [
    {url: "/home", userAgent: "Mozilla/5.0...", referrer: "google.com
", timestamp: "2024-01-01 10:15"},
    {url: "/products", userAgent: "Chrome/96...", referrer: "/home",
timestamp: "2024-01-01 10:16"},
    ...
]

for entry in logs do
    analyzeWebLog(analyzer, entry)

stats = getWebStats(analyzer)
// Результат: топ-100 сторінок, топ-10 браузерів, топ-50 referrer'ів
```

Приклад 3: Виявлення DDoS атак

Моніторинг мережевого трафіку для виявлення атак:

```
struct DDoSDetector {
    ipCounter: LossyCounting
    portCounter: LossyCounting
    packetSizeDistribution: LossyCounting
    suspiciousIPs: set of IP
    alertThreshold: float
}

function detectDDoS(detector: DDoSDetector, packet: NetworkPacket)
    // Обробляємо IP-адресу джерела
    processElement(detector.ipCounter, packet.sourceIP)

    // Обробляємо порт призначення
```

```
    processElement(detector.portCounter, packet.destPort)

    // Аналізуємо розмір пакета
    sizeCategory = categorizePacketSize(packet.size)
    processElement(detector.packetSizeDistribution, sizeCategory)

    // Перевіряємо на аномалії
    if detector.ipCounter.N mod 1000 == 0 then
        checkForAnomalies(detector)

function checkForAnomalies(detector: DDoSDetector)
    totalPackets = detector.ipCounter.N

    // Знаходимо IP з надмірним трафіком
    frequentIPs = getFrequentItems(detector.ipCounter, detector.alert
Threshold)

    for (ip, frequency) in frequentIPs do
        percentageTraffic = frequency / totalPackets

        if percentageTraffic > 0.1 then  // >10% всього трафіку
            detector.suspiciousIPs.add(ip)
            raiseAlert("Potential DDoS from IP: " + ip, percentageTra
ffic)

    // Аналізуємо розподіл портів
    portDistribution = getFrequentItems(detector.portCounter, 0.05)
    if len(portDistribution) == 1 and portDistribution[0].frequency >
0.9 * totalPackets then
        raiseAlert("Single port attack detected", portDistribution[0]
)

// Використання
detector = DDoSDetector(
    ipCounter = initLossyCounting(0.01, 0.001),  // IP з >1% трафіку
    portCounter = initLossyCounting(0.05, 0.005),
    packetSizeDistribution = initLossyCounting(0.1, 0.01),
    suspiciousIPs = set(),
    alertThreshold = 0.01
)

// Симуляція атаки
normalTraffic = generateNormalTraffic(1000)
attackTraffic = generateDDoSTraffic(9000, attackerIP="192.168.1.100")
```

```
for packet in interleave(normalTraffic, attackTraffic) do
    detectDDoS(detector, packet)

// Результат: Alert - IP 192.168.1.100 генерує 90% трафіку
```
Приклад 4: Аналіз соціальних мереж

Знаходження трендових хештегів:

```
struct TrendingHashtags {
    globalTrends: LossyCounting
    languageTrends: HashMap<string, LossyCounting>
    timePeriods: SlidingWindow<LossyCounting>
    minHashtagLength: integer
}

function processTweet(trending: TrendingHashtags, tweet: Tweet)
    hashtags = extractHashtags(tweet.text)
    language = detectLanguage(tweet.text)

    for hashtag in hashtags do
        if len(hashtag) >= trending.minHashtagLength then
            // Глобальні тренди
            processElement(trending.globalTrends, hashtag.toLowerCase
())

            // Тренди по мовах
            if language not in trending.languageTrends then
                trending.languageTrends[language] = initLossyCounting
(0.001, 0.0001)
            processElement(trending.languageTrends[language], hashtag
)

            // Часові тренди
            currentWindow = trending.timePeriods.getCurrentWindow()
            processElement(currentWindow, hashtag)

function getTrendingNow(trending: TrendingHashtags, topK: integer = 1
0): list
    currentTrends = getFrequentItems(trending.timePeriods.getCurrentW
indow())

    // Обчислюємо швидкість зростання
    trendScores = []
    for (hashtag, currentFreq) in currentTrends do
        previousFreq = getPreviousFrequency(trending, hashtag)
        growthRate = (currentFreq - previousFreq) / max(previousFreq,
```

```
1)

        trendScores.append({
            hashtag: hashtag,
            frequency: currentFreq,
            growth: growthRate,
            score: currentFreq * (1 + growthRate)
        })

    // Сортуємо за комбінованим score
    trendScores.sortByDescending(t => t.score)

    return trendScores[0:topK]

// Приклад
trending = TrendingHashtags(
    globalTrends = initLossyCounting(0.0001, 0.00001),  // топ 0.01%
    languageTrends = HashMap(),
    timePeriods = SlidingWindow(windowSize=3600, slideInterval=300),
// 1 година, оновлення кожні 5 хв
    minHashtagLength = 3
)

tweets = streamTweets()
for tweet in tweets do
    processTweet(trending, tweet)

    if processed % 10000 == 0 then
        currentTrends = getTrendingNow(trending)
        updateDashboard(currentTrends)
```

Приклад 5: Оптимізація бази даних

Аналіз частих запитів для створення індексів:

```
struct QueryAnalyzer {
    queryPatterns: LossyCounting
    tableAccess: HashMap<string, LossyCounting>
    joinPatterns: LossyCounting
    slowQueries: PriorityQueue<(query, frequency, avgTime)>
}

function analyzeQuery(analyzer: QueryAnalyzer, query: SQLQuery, execu
tionTime: float)
    // Нормалізуємо запит (замінюємо константи на placeholders)
    pattern = normalizeQuery(query)
    processElement(analyzer.queryPatterns, pattern)
```

```
    // Аналізуємо доступ до таблиць
    tables = extractTables(query)
    for table in tables do
        if table not in analyzer.tableAccess then
            analyzer.tableAccess[table] = initLossyCounting(0.01, 0.0
01)

        columns = extractColumns(query, table)
        for column in columns do
            processElement(analyzer.tableAccess[table], column)

    // Аналізуємо JOIN патерни
    joins = extractJoins(query)
    for join in joins do
        processElement(analyzer.joinPatterns, join)

    // Відстежуємо повільні запити
    if executionTime > SLOW_QUERY_THRESHOLD then
        analyzer.slowQueries.add((pattern, 1, executionTime))

function suggestOptimizations(analyzer: QueryAnalyzer): list of Optim
ization
    optimizations = []

    // Пропонуємо індекси для частих колонок
    for (table, columnCounter) in analyzer.tableAccess do
        frequentColumns = getFrequentItems(columnCounter, 0.05)  // к
олонки в >5% запитів

        for (column, frequency) in frequentColumns do
            if not hasIndex(table, column) then
                benefit = estimateIndexBenefit(table, column, frequen
cy)
                optimizations.append(IndexSuggestion(table, column, b
enefit))

    // Пропонуємо матеріалізовані представлення для частих JOIN
    frequentJoins = getFrequentItems(analyzer.joinPatterns, 0.02)
    for (joinPattern, frequency) in frequentJoins do
        if isCostlyJoin(joinPattern) then
            optimizations.append(MaterializedViewSuggestion(joinPatte
rn, frequency))

    // Аналізуємо повільні часті запити
```

```
    slowFrequent = analyzer.slowQueries.getTopK(20)
    for (query, freq, avgTime) in slowFrequent do
        if freq > 100 and avgTime > 1.0 then  // >100 разів, >1 сек
            optimizations.append(QueryRewriteSuggestion(query, avgTim
e))

    return optimizations.sortByDescending(o => o.estimatedBenefit)
```

Переваги та недоліки

Переваги

- Детерміністичні гарантії: На відміну від імовірнісних алгоритмів, гарантує знаходження всіх частих елементів.

- Проста реалізація: Алгоритм простий для розуміння та реалізації порівняно з іншими методами.

- Контрольована похибка: Максимальна похибка обмежена параметром ε.

- Ефективне використання пам’яті: Використовує O(1/ε log εN) пам’яті в гіршому випадку.

- Підходить для потоків: Однопрохідний алгоритм, ідеальний для обробки потоків даних.

- Немає хеш-колізій: На відміну від sketch-алгоритмів, зберігає точні елементи.

Недоліки

- Може включати помилкові елементи: Елементи з частотою трохи нижче порогу можуть потрапити в результат.

- Використання пам’яті залежить від розподілу: Для рівномірних розподілів може використовувати більше пам’яті.

- Необхідність вибору параметрів: Потрібно заздалегідь визначити ε та s.

- Періодичні сплески обчислень: Очищення в кінці вікна може створювати затримки.

- Не підходить для малих потоків: Накладні витрати можуть бути значними для малих датасетів.

- Складність оцінки точної частоти: Зберігається лише нижня межа частоти.

Контрольні запитання

1. Які основні параметри алгоритму Lossy Counting та як вони впливають на його роботу?

2. Що таке bucket у контексті Lossy Counting та яка його роль?

3. Поясніть значення delta (Δ) для кожного елемента та як воно використовується.

4. Які гарантії надає Lossy Counting щодо частих елементів?

5. Як працює процес очищення в кінці кожного вікна?

6. Чому розмір вікна дорівнює ceil(1/ε)?

7. Як Lossy Counting відрізняється від імовірнісних алгоритмів типу Count-Min Sketch?

8. Яка теоретична оцінка використання пам’яті алгоритмом?

9. Як можна адаптувати Lossy Counting для знаходження top-k елементів?

10. Які основні сценарії використання Lossy Counting у реальних системах?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму Lossy Counting для аналізу потоків даних

2. Мета

Вивчити принципи детермінованого підрахунку частих елементів у потоках, реалізувати алгоритм Lossy Counting, дослідити вплив параметрів на точність та використання пам’яті, порівняти з іншими алгоритмами для потоків.

3. Завдання

Реалізувати базовий алгоритм Lossy Counting, створити модифікації для різних сценаріїв використання, провести експерименти з реальними та синтетичними потоками даних, розробити візуалізацію роботи алгоритму.

4. Опис

Лабораторна робота передбачає реалізацію Lossy Counting з підтримкою різних типів даних, дослідження поведінки при різних розподілах вхідних даних, створення розширень для ієрархічних даних та паралельної обробки.

5. Порядок виконання

- Реалізувати базову структуру даних LossyCounting

- Створити функції processElement та cleanupWindow

- Реалізувати функцію getFrequentItems з різними порогами
- Додати підтримку різних типів даних (рядки, числа, об’єкти)
- Створити генератори тестових потоків з різними розподілами
- Реалізувати візуалізацію внутрішнього стану
- Провести експерименти з параметрами ε та s
- Порівняти з Count-Min Sketch та Space-Saving
- Реалізувати паралельну версію
- Створити демонстраційний застосунок

6. Результат

Працююча реалізація Lossy Counting з графічним інтерфейсом, набір тестів для перевірки коректності, графіки залежності точності та пам’яті від параметрів, порівняльний аналіз з іншими алгоритмами, рекомендації щодо вибору параметрів для різних застосувань.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
- Клас LossyCounting з усіма операціями
- Генератори потоків даних з різними розподілами
- Модуль візуалізації стану алгоритму
- Реалізація модифікацій (top-k, ієрархічна, паралельна)
- Порівняльні тести з іншими алгоритмами

2. Продемонструвати роботу додатку:
- Обробка потоку з відображенням поточного стану
- Знаходження частих елементів з різними порогами
- Вплив параметрів на результат та пам’ять
- Робота з різними типами даних
- Порівняння результатів різних алгоритмів

3. Бути готовим відповісти на питання:
- Як вибрати оптимальні параметри для конкретної задачі?
- Чому важливо видаляти елементи в кінці вікна?

- Коли Lossy Counting кращий за імовірнісні методи?
- Як масштабувати алгоритм для великих потоків?

Тести

1. Що означає параметр ε в Lossy Counting?
- 1) Поріг частоти
- 2) Максимальна похибка
- 3) Розмір вікна
- 4) Кількість елементів

2. Коли видаляються елементи в Lossy Counting?
- 1) При кожному новому елементі
- 2) В кінці кожного вікна
- 3) Коли закінчується пам’ять
- 4) Випадково

3. Яка умова для видалення елемента з частотою f та delta Δ?
- 1) $f < \varepsilon$
- 2) $f + \Delta \leqslant$ поточний_bucket
- 3) $f > s$
- 4) $\Delta > \varepsilon$

4. Який розмір вікна w при $\varepsilon = 0.01$?
- 1) 10
- 2) 100
- 3) 1000
- 4) 0.01

5. Lossy Counting гарантує знаходження елементів з частотою:
- 1) $\geqslant \varepsilon N$
- 2) $\geqslant (s-\varepsilon)N$
- 3) $\geqslant sN$
- 4) $\geqslant N/w$

Відповіді:

1-2,

2-2,

3-2,

4-2,

5-3

Тести підвищеної складності

1. Чому delta нового елемента дорівнює bucketId - 1?

- 1) Для економії пам’яті

- 2) Це максимальна можлива частота елемента до поточного моменту

- 3) Випадкове значення

- 4) Для швидкості обчислень

2. Яка максимальна кількість елементів може зберігатися в структурі?

- 1) 1/ε

- 2) 1/s

- 3) (1/ε)log(εN)

- 4) N

3. Що станеться з елементом, який з’являється рівно один раз у кожному вікні?

- 1) Завжди зберігається

- 2) Завжди видаляється в кінці вікна

- 3) Залежить від розміру вікна

- 4) Накопичує частоту

4. Як Lossy Counting обробляє дублікати?

- 1) Зберігає всі копії

- 2) Збільшує лічильник частоти

- 3) Ігнорує дублікати

- 4) Створює новий запис

5. Чому Lossy Counting називається “lossy”?

- 1) Втрачає точність

- 2) Втрачає (видаляє) рідкісні елементи

- 3) Втрачає порядок елементів

- 4) Втрачає час

6. Яка складність операції запиту частих елементів?

- 1) O(1)

- 2) O(log n)

- 3) O(|D|), де |D| - розмір структури

- 4) O(N)

7. Коли Lossy Counting використовує найбільше пам’яті?

- 1) На початку потоку

- 2) При рівномірному розподілі

- 3) При skewed розподілі

- 4) В кінці потоку

8. Що відбувається при s = ε?

- 1) Алгоритм не працює

- 2) Максимальна точність

- 3) Немає гарантій щодо результату

- 4) Мінімальне використання пам’яті

9. Як змінити Lossy Counting для підтримки зважених елементів?

- 1) Неможливо

- 2) Збільшувати frequency на вагу замість 1

- 3) Використовувати окрему структуру

- 4) Змінити розмір вікна

10. Яка основна перевага над Space-Saving алгоритмом?

- 1) Менше пам’яті

- 2) Детерміністичні гарантії повноти

- 3) Швидша робота

- 4) Простіша реалізація

Відповіді:

1. 2) Елемент міг з’явитися в будь-який момент поточного bucket, тому максимальна невідома частота = bucketId-1

2. 3) Теоретична верхня межа кількості елементів у структурі

3. 2) f=1, Δ=bucketId-1, в кінці вікна 1+(bucketId-1)⩽bucketId, тому видаляється

4. 2) Збільшується лічильник для існуючого елемента

5. 2) “Втрачає” рідкісні елементи для економії пам’яті

6. 3) Потрібно перевірити всі збережені елементи

7. 2) Коли багато різних елементів з близькими частотами

8. 3) Порушується умова ε < s, алгоритм може працювати некоректно

9. 2) Замість increment на 1, додавати вагу елемента

10. 2) Гарантує знаходження всіх елементів з частотою ⩾ sN

Завдання для самостійної роботи

1. Реалізуйте Lossy Counting з підтримкою sliding window для аналізу останніх N елементів.

2. Створіть розподілену версію Lossy Counting для обробки на кількох серверах.

3. Розробіть адаптивну версію, що автоматично налаштовує ε базуючись на доступній пам’яті.

4. Реалізуйте Sticky Sampling алгоритм та порівняйте з Lossy Counting.

5. Створіть версію для обробки багатовимірних даних (наприклад, пари користувач-товар).

6. Розробіть механізм персистентності для збереження та відновлення стану.

7. Реалізуйте інкрементальну версію для оновлюваних потоків.

8. Створіть візуалізацію еволюції частих елементів у часі.

9. Розробіть версію з підтримкою регулярних виразів для текстових потоків.

10. Дослідіть застосування Lossy Counting для виявлення аномалій у часових рядах.

Теми лабораторних робіт

1. Базова реалізація Lossy Counting та дослідження параметрів.

2. Порівняльний аналіз Lossy Counting, Count-Min Sketch та Space-Saving.

3. Розробка системи моніторингу веб-трафіку на основі Lossy Counting.

4. Створення детектора DDoS-атак з використанням потокових алгоритмів.

5. Реалізація системи рекомендацій з підрахунком популярних товарів.

6. Аналіз логів розподілених систем для виявлення проблем.

7. Розробка системи трендів для соціальних мереж.

8. Створення оптимізатора запитів бази даних на основі аналізу патернів.

9. Реалізація системи виявлення шахрайства у фінансових транзакціях.

10. Дослідження застосування Lossy Counting для аналізу IoT даних.

Space-Saving

Теоретичні відомості

Алгоритм Space-Saving є одним з найефективніших детермінованих алгоритмів для знаходження частих елементів у потоках даних. Розроблений Ахмедом Метвалі, Дівьякантом Агравалом та Амром Ель Аббаді у 2005 році, цей алгоритм використовує фіксований обсяг пам'яті для відстеження k найчастіших елементів і гарантує, що жоден справді частий елемент не буде пропущений.

Space-Saving зберігає фіксовану кількість лічильників для відстеження елементів та їх частот. Коли надходить новий елемент, алгоритм або збільшує його лічильник (якщо елемент вже відстежується), або замінює елемент з найменшою частотою. Ключова інновація полягає в тому, що при заміні новий елемент отримує частоту замінюваного елемента плюс один, що забезпечує важливі теоретичні гарантії.

Алгоритм працює так:

Він підтримує в пам'яті список фіксованого розміру `k`, де `k` — це кількість елементів, які ми хочемо відстежувати. Кожен запис у списку — це пара `(елемент, його частота)`.

Коли з потоку надходить новий елемент:

1. Якщо елемент вже відстежується: Його лічильник частоти просто збільшується на 1.

2. Якщо елемент новий, а у списку є вільне місце: Він додається до списку з частотою 1.

3. Якщо елемент новий, а список заповнений (найцікавіший випадок):

- Знаходиться елемент у списку, у якого найменша частота.

- Цей старий елемент викидається зі списку.

- На його місце ставиться новий елемент, а його частота встановлюється на (частота видаленого елемента + 1).

Замість того, щоб періодично зменшувати всі лічильники, як у Lossy Counting, Space-Saving завжди замінює елемент із найменшою частотою.

Такий підхід гарантує, що справжні “важковаговики”, які з’являються часто, швидко накопичать велике значення лічильника і їх буде дуже “важко” витіснити зі списку. При цьому частота, що присвоюється новому елементу, слугує наближеною оцінкою його реальної частоти, що дозволяє зберігати високу точність.

Алгоритм підтримує два основні інваріанти. По-перше, сума всіх лічильників завжди дорівнює загальній кількості оброблених елементів. По-друге, для кожного елемента, що не зберігається в структурі, його справжня частота не перевищує мінімальну частоту серед збережених елементів.

Для ефективної реалізації Space-Saving використовує комбінацію кількох структур даних. Основна структура містить хеш-таблицю для швидкого доступу до елементів, двозв’язний список для організації елементів за частотами та допоміжні структури для оптимізації операцій.

```
struct SpaceSaving {
    k: integer                    // максимальна кількість елементів
для відстеження
    n: integer                    // загальна кількість оброблених ел
ементів
    counters: HashMap<item, Counter>  // відображення елемент -> лічи
льник
    buckets: HashMap<count, Bucket>   // відображення частота -> buck
et
    minCount: integer             // мінімальна частота серед всіх лі
чильників
}

struct Counter {
    item: any           // елемент
    count: integer      // оцінена частота
    error: integer      // максимальна переоцінка
    bucket: Bucket      // посилання на bucket з цією частотою
}

struct Bucket {
    count: integer              // частота для цього bucket
```

```
    counters: DoublyLinkedList  // список лічильників з цією частотою
    prev: Bucket                // попередній bucket (менша частота)
    next: Bucket                // наступний bucket (більша частота)
}
```

Ініціалізація алгоритму

Процес ініціалізації створює порожню структуру даних з заданою місткістю k. На початку роботи алгоритм не містить жодних елементів, і всі структури даних порожні.

```
function initSpaceSaving(k: integer): SpaceSaving
    if k <= 0 then
        throw Error("k must be positive")

    ss = SpaceSaving(
        k = k,
        n = 0,
        counters = new HashMap(),
        buckets = new HashMap(),
        minCount = 0
    )

    return ss
```

Операція PROCESS (обробка елемента)

Операція обробки нового елемента є центральною для алгоритму. Коли надходить новий елемент, алгоритм спочатку перевіряє, чи вже відстежується цей елемент. Якщо так, його лічильник збільшується. Якщо ні, і є вільне місце, елемент додається з частотою 1. Якщо структура заповнена, елемент з найменшою частотою замінюється новим елементом.

```
function processElement(ss: SpaceSaving, item: any)
    ss.n += 1

    if item in ss.counters then
        // Елемент вже відстежується, збільшуємо лічильник
        incrementCounter(ss, item)
    else if ss.counters.size() < ss.k then
        // Є вільне місце, додаємо новий елемент
        addNewCounter(ss, item)
    else
        // Структура заповнена, замінюємо елемент з мінімальною часто
тою
        replaceMinCounter(ss, item)

function incrementCounter(ss: SpaceSaving, item: any)
```

```
    counter = ss.counters[item]
    oldCount = counter.count
    newCount = oldCount + 1

    // Видаляємо лічильник зі старого bucket
    removeCounterFromBucket(ss, counter)

    // Оновлюємо частоту
    counter.count = newCount

    // Додаємо до нового bucket
    addCounterToBucket(ss, counter, newCount)

function addNewCounter(ss: SpaceSaving, item: any)
    // Створюємо новий лічильник з частотою 1
    counter = Counter(
        item = item,
        count = 1,
        error = 0,
        bucket = null
    )

    // Додаємо до структури
    ss.counters[item] = counter
    addCounterToBucket(ss, counter, 1)

    // Оновлюємо мінімальну частоту
    if ss.minCount == 0 or ss.minCount > 1 then
        ss.minCount = 1

function replaceMinCounter(ss: SpaceSaving, item: any)
    // Знаходимо bucket з мінімальною частотою
    minBucket = ss.buckets[ss.minCount]

    // Беремо будь-який лічильник з цього bucket
    oldCounter = minBucket.counters.getFirst()
    oldItem = oldCounter.item
    oldCount = oldCounter.count

    // Видаляємо старий елемент
    removeCounterFromBucket(ss, oldCounter)
    ss.counters.remove(oldItem)

    // Створюємо новий лічильник з частотою oldCount + 1
    newCounter = Counter(
```

```
        item = item,
        count = oldCount + 1,
        error = oldCount,  // максимальна переоцінка
        bucket = null
    )

    // Додаємо новий елемент
    ss.counters[item] = newCounter
    addCounterToBucket(ss, newCounter, oldCount + 1)
```

Управління buckets

Buckets організовані як двозв'язний список, впорядкований за частотами. Це дозволяє ефективно знаходити bucket з мінімальною частотою та переміщати лічильники між buckets при зміні їх частот.

```
function addCounterToBucket(ss: SpaceSaving, counter: Counter, count:
integer)
    if count not in ss.buckets then
        // Створюємо новий bucket
        newBucket = Bucket(
            count = count,
            counters = new DoublyLinkedList(),
            prev = null,
            next = null
        )

        // Вставляємо bucket у впорядковану послідовність
        insertBucket(ss, newBucket)
        ss.buckets[count] = newBucket

    // Додаємо лічильник до bucket
    bucket = ss.buckets[count]
    bucket.counters.append(counter)
    counter.bucket = bucket

    // Оновлюємо мінімальну частоту якщо потрібно
    updateMinCount(ss)

function removeCounterFromBucket(ss: SpaceSaving, counter: Counter)
    bucket = counter.bucket
    bucket.counters.remove(counter)

    // Якщо bucket порожній, видаляємо його
    if bucket.counters.isEmpty() then
        removeBucket(ss, bucket)
```

```
        ss.buckets.remove(bucket.count)

    counter.bucket = null

function insertBucket(ss: SpaceSaving, newBucket: Bucket)
    // Знаходимо правильну позицію для вставки
    count = newBucket.count

    // Знаходимо сусідні buckets
    prevBucket = null
    nextBucket = null

    for (c, b) in ss.buckets do
        if c < count and (prevBucket == null or c > prevBucket.count)
then
            prevBucket = b
        if c > count and (nextBucket == null or c < nextBucket.count)
then
            nextBucket = b

    // Вставляємо між prevBucket та nextBucket
    newBucket.prev = prevBucket
    newBucket.next = nextBucket

    if prevBucket != null then
        prevBucket.next = newBucket
    if nextBucket != null then
        nextBucket.prev = newBucket

function updateMinCount(ss: SpaceSaving)
    if ss.buckets.isEmpty() then
        ss.minCount = 0
    else
        // Знаходимо bucket з найменшою частотою
        minBucket = null
        for (count, bucket) in ss.buckets do
            if minBucket == null or count < minBucket.count then
                minBucket = bucket
        ss.minCount = minBucket.count
```

Операція QUERY (запит елементів)

Запит повертає елементи з частотою вище заданого порогу або top-k елементів. Важливо враховувати похибку при фільтрації результатів.

```
function getFrequentItems(ss: SpaceSaving, threshold: integer): list
of (item, count, error)
```

```
    results = []

    for (item, counter) in ss.counters do
        // Гарантована мінімальна частота = count - error
        guaranteedCount = counter.count - counter.error

        if guaranteedCount >= threshold then
            results.append((item, counter.count, counter.error))

    // Сортуємо за оціненою частотою
    results.sortByDescending(r => r.count)

    return results

function getTopK(ss: SpaceSaving, k: integer): list of (item, count,
error)
    results = []

    // Збираємо всі елементи
    for (item, counter) in ss.counters do
        results.append((item, counter.count, counter.error))

    // Сортуємо за частотою
    results.sortByDescending(r => r.count)

    // Повертаємо перші k
    return results[0:min(k, results.length)]
```

Space-Saving надає важливі теоретичні гарантії щодо точності оцінок:

1. **Гарантія для частих елементів**: Всі елементи з справжньою частотою $f_i > n/k$ гарантовано будуть у структурі

2. **Межа похибки**: Для кожного елемента максимальна переоцінка не перевищує n/k

3. **Точність для top-k**: Якщо елемент входить у справжні top-k, він буде у результаті

Модифікації та розширення

1. Space-Saving з вікнами

Адаптація для аналізу даних у ковзному вікні:

```
struct WindowedSpaceSaving {
    windowSize: integer
    currentWindow: SpaceSaving
    timestamps: CircularBuffer<(item, timestamp)>
```

```
    lastCleanup: timestamp
}

function processWindowedElement(wss: WindowedSpaceSaving, item: any,
timestamp: integer)
    // Додаємо до поточного вікна
    processElement(wss.currentWindow, item)

    // Зберігаємо для майбутнього видалення
    wss.timestamps.add((item, timestamp))

    // Видаляємо старі елементи
    while wss.timestamps.size() > wss.windowSize do
        oldEntry = wss.timestamps.removeFirst()
        decrementCounter(wss.currentWindow, oldEntry.item)
```

2. Ієрархічний Space-Saving

Для аналізу даних на різних рівнях деталізації:

```
struct HierarchicalSpaceSaving {
    levels: array of SpaceSaving
    aggregationFunctions: array of function
}

function processHierarchicalElement(hss: HierarchicalSpaceSaving, ite
m: any)
    // Обробляємо на всіх рівнях ієрархії
    for i = 0 to hss.levels.length - 1 do
        aggregatedItem = hss.aggregationFunctions[i](item)
        processElement(hss.levels[i], aggregatedItem)
```

Область застосування

Space-Saving алгоритм широко використовується в системах, де потрібно ефективно відстежувати найчастіші елементи:

- Мережевий моніторинг: Виявлення найактивніших IP-адрес, відстеження популярних доменів, аналіз протоколів.

- Кешування: Визначення найпопулярніших об’єктів для оптимізації кеш-пам’яті.

- Рекомендаційні системи: Відстеження популярних товарів, фільмів, пісень для персоналізованих рекомендацій.

- Аналіз запитів: Знаходження частих SQL-запитів для оптимізації бази даних.

- Обробка тексту: Виявлення найчастіших слів, фраз, n-грам у великих текстових корпусах.

- Системи безпеки: Виявлення аномальної активності через відстеження частих подій.

Приклади застосування

Приклад 1: Базова робота алгоритму

Розглянемо роботу Space-Saving з k=3 для потоку елементів:

```
Потік: A, B, A, C, A, B, D, A, E, B, C, A

Ініціалізація: k=3, порожня структура

Крок 1: A
- Додаємо A з count=1, error=0
- Стан: {A:1}

Крок 2: B
- Додаємо B з count=1, error=0
- Стан: {A:1, B:1}

Крок 3: A
- Збільшуємо A до count=2
- Стан: {A:2, B:1}

Крок 4: C
- Додаємо C з count=1, error=0
- Стан: {A:2, B:1, C:1}

Крок 5: A
- Збільшуємо A до count=3
- Стан: {A:3, B:1, C:1}

Крок 6: B
- Збільшуємо B до count=2
- Стан: {A:3, B:2, C:1}

Крок 7: D (структура заповнена)
- Замінюємо C (min=1) на D
- D отримує count=2 (1+1), error=1
- Стан: {A:3, B:2, D:2(e=1)}

Крок 8: A
- Збільшуємо A до count=4
```

```
- Стан: {A:4, B:2, D:2(e=1)}

Крок 9: E
- Замінюємо B або D (обидва з count=2)
- Нехай замінюємо D
- E отримує count=3, error=2
- Стан: {A:4, B:2, E:3(e=2)}

Крок 10: B
- Збільшуємо B до count=3
- Стан: {A:4, B:3, E:3(e=2)}

Крок 11: C
- Замінюємо E (хоча count=3, але це найменше)
- C отримує count=4, error=3
- Стан: {A:4, B:3, C:4(e=3)}

Крок 12: A
- Збільшуємо A до count=5
- Фінальний стан: {A:5, B:3, C:4(e=3)}

Справжні частоти: A=5, B=3, C=2, D=1, E=1
Space-Saving зберіг A і B правильно, переоцінив C
```

Приклад 2: Моніторинг веб-трафіку

Система для відстеження найпопулярніших URL:

```
struct WebTrafficMonitor {
    urlTracker: SpaceSaving
    userTracker: SpaceSaving
    userUrlPairs: SpaceSaving
    hourlyTrackers: array[24] of SpaceSaving
}

function monitorRequest(monitor: WebTrafficMonitor, request: HTTPRequ
est)
    url = normalizeURL(request.url)
    userId = request.userId
    hour = request.timestamp.getHour()

    // Відстежуємо популярні URL
    processElement(monitor.urlTracker, url)

    // Відстежуємо активних користувачів
    processElement(monitor.userTracker, userId)
```

```
    // Відстежуємо пари користувач-URL
    pair = userId + ":" + url
    processElement(monitor.userUrlPairs, pair)

    // Погодинна статистика
    processElement(monitor.hourlyTrackers[hour], url)

function getTrafficReport(monitor: WebTrafficMonitor): TrafficReport
    return TrafficReport(
        topURLs = getTopK(monitor.urlTracker, 100),
        topUsers = getTopK(monitor.userTracker, 50),
        userPatterns = analyzeUserPatterns(monitor.userUrlPairs),
        peakHours = findPeakTrafficHours(monitor.hourlyTrackers)
    )

// Використання
monitor = WebTrafficMonitor(
    urlTracker = initSpaceSaving(1000),      // top-1000 URLs
    userTracker = initSpaceSaving(500),      // top-500 users
    userUrlPairs = initSpaceSaving(5000),    // top-5000 pairs
    hourlyTrackers = [initSpaceSaving(100) for i in 0..23]
)

// Обробка запитів
requests = streamHTTPRequests()
for request in requests do
    monitorRequest(monitor, request)

    if processed % 10000 == 0 then
        report = getTrafficReport(monitor)
        updateDashboard(report)
```

Приклад 3: Система кешування

Використання Space-Saving для адаптивного кешування:

```
struct AdaptiveCache {
    cache: HashMap<key, value>
    popularity: SpaceSaving
    maxCacheSize: integer
    evictionThreshold: float
}

function get(cache: AdaptiveCache, key: string): value
    // Оновлюємо популярність
    processElement(cache.popularity, key)
```

```
    if key in cache.cache then
        return cache.cache[key]
    else
        // Cache miss
        value = fetchFromBackend(key)

        // Вирішуємо чи кешувати
        if shouldCache(cache, key) then
            ensureSpace(cache)
            cache.cache[key] = value

        return value

function shouldCache(cache: AdaptiveCache, key: string): boolean
    // Перевіряємо чи елемент достатньо популярний
    itemInfo = getItemInfo(cache.popularity, key)

    if itemInfo == null then
        return false

    // Кешуємо якщо гарантована частота вище порогу
    guaranteedFreq = itemInfo.count - itemInfo.error
    totalRequests = cache.popularity.n

    return guaranteedFreq / totalRequests > cache.evictionThreshold

function ensureSpace(cache: AdaptiveCache)
    while cache.cache.size() >= cache.maxCacheSize do
        // Знаходимо найменш популярний елемент у кеші
        leastPopular = findLeastPopularCached(cache)
        cache.cache.remove(leastPopular)

function findLeastPopularCached(cache: AdaptiveCache): key
    minScore = INFINITY
    victimKey = null

    for key in cache.cache.keys() do
        itemInfo = getItemInfo(cache.popularity, key)
        if itemInfo != null then
            score = itemInfo.count
        else
            score = 0  // не відстежується = дуже непопулярний

        if score < minScore then
            minScore = score
```

```
            victimKey = key

    return victimKey
```

Приклад 4: Виявлення трендів

Система для виявлення елементів зі зростаючою популярністю:

```
struct TrendDetector {
    currentPeriod: SpaceSaving
    previousPeriod: SpaceSaving
    periodDuration: integer
    lastSwitch: timestamp
}

function processTrendItem(detector: TrendDetector, item: any, timesta
mp: integer)
    // Перевіряємо чи потрібно переключити періоди
    if timestamp - detector.lastSwitch >= detector.periodDuration the
n
        switchPeriods(detector, timestamp)

    // Обробляємо в поточному періоді
    processElement(detector.currentPeriod, item)

function switchPeriods(detector: TrendDetector, timestamp: integer)
    // Зберігаємо поточний як попередній
    detector.previousPeriod = detector.currentPeriod

    // Створюємо новий поточний період
    detector.currentPeriod = initSpaceSaving(detector.currentPeriod.k
)
    detector.lastSwitch = timestamp

function detectTrends(detector: TrendDetector): list of TrendInfo
    trends = []

    currentItems = getAllItems(detector.currentPeriod)

    for (item, currentCount, currentError) in currentItems do
        previousInfo = getItemInfo(detector.previousPeriod, item)

        if previousInfo != null then
            previousCount = previousInfo.count

            // Обчислюємо зростання
            growth = (currentCount - previousCount) / max(previousCou
```

```
nt, 1)

            // Враховуємо похибки
            minGrowth = ((currentCount - currentError) - (previousCou
nt + previousInfo.error)) /
                        max(previousCount + previousInfo.error, 1)

            if minGrowth > 0.5 then  // зростання мінімум на 50%
                trends.append(TrendInfo(
                    item = item,
                    currentCount = currentCount,
                    previousCount = previousCount,
                    growthRate = growth,
                    confidence = minGrowth
                ))
        else
            // Новий елемент у поточному періоді
            if currentCount - currentError > detector.currentPeriod.n
/ detector.currentPeriod.k then
                trends.append(TrendInfo(
                    item = item,
                    currentCount = currentCount,
                    previousCount = 0,
                    growthRate = INFINITY,
                    confidence = 1.0
                ))

    return trends.sortByDescending(t => t.growthRate)
```

Приклад 5: Розподілена версія

Space-Saving для розподілених систем:

```
struct DistributedSpaceSaving {
    nodeId: string
    local: SpaceSaving
    mergeInterval: integer
    coordinator: CoordinatorNode
}

function processDistributed(dss: DistributedSpaceSaving, items: strea
m)
    for item in items do
        processElement(dss.local, item)

        if dss.local.n % dss.mergeInterval == 0 then
            sendLocalState(dss)
```

```
function sendLocalState(dss: DistributedSpaceSaving)
    // Створюємо знімок локального стану
    snapshot = createSnapshot(dss.local)

    // Відправляємо координатору
    dss.coordinator.receiveSnapshot(dss.nodeId, snapshot)

// На координаторі
struct CoordinatorNode {
    nodeSnapshots: HashMap<nodeId, SpaceSavingSnapshot>
    globalView: SpaceSaving
}

function mergeSnapshots(coordinator: CoordinatorNode)
    // Збираємо всі елементи з усіх вузлів
    allItems = []

    for (nodeId, snapshot) in coordinator.nodeSnapshots do
        for (item, counter) in snapshot.counters do
            allItems.append((item, counter.count, counter.error, node
Id))

    // Групуємо за елементами
    itemGroups = groupBy(allItems, item => item[0])

    // Створюємо глобальний Space-Saving
    globalK = coordinator.globalView.k
    globalSS = initSpaceSaving(globalK)

    // Обробляємо кожен унікальний елемент
    for (item, occurrences) in itemGroups do
        totalCount = sum(occurrences.map(o => o[1]))
        maxError = max(occurrences.map(o => o[2]))

        // Емулюємо додавання елемента totalCount разів
        for i = 1 to min(totalCount, globalK) do
            processElement(globalSS, item)

    coordinator.globalView = globalSS
```

Переваги та недоліки

Переваги

- Оптимальне використання пам’яті: Використовує рівно k лічильників для відстеження top-k елементів.

- Детерміністичні гарантії: Гарантує, що всі елементи з частотою > n/k будуть збережені.

- Простота реалізації: Алгоритм відносно простий порівняно з іншими методами.

- Точні лічильники: На відміну від sketch-алгоритмів, зберігає точні значення для відстежуваних елементів.

- Швидкі операції: Амортизована складність O(1) для обробки елемента.

- Підходить для top-k запитів: Ідеально підходить для знаходження k найчастіших елементів.

Недоліки

- Фіксований розмір: Кількість відстежуваних елементів фіксована заздалегідь.

- Переоцінка частот: Може переоцінювати частоти елементів через заміни.

- Не підходить для рідкісних елементів: Погано працює для відстеження елементів з низькими частотами.

- Складність точної реалізації: Ефективна реалізація вимагає складних структур даних.

- Неможливість видалення: Немає прямого способу зменшити лічильник елемента.

- Залежність від порядку: Результат може залежати від порядку надходження елементів.

Контрольні запитання

1. Яка основна ідея алгоритму Space-Saving для економії пам’яті?

2. Що означає поле error у структурі Counter та як воно використовується?

3. Які гарантії надає Space-Saving щодо елементів з частотою вище n/k?

4. Як працює заміна елемента при заповненій структурі даних?

5. Чому новий елемент отримує частоту замінюваного елемента плюс один?

6. Як організовані buckets та яка їх роль в алгоритмі?

7. Які основні відмінності між Space-Saving та Lossy Counting?

8. Як можна адаптувати Space-Saving для роботи з вікнами?

9. Яка складність операцій в Space-Saving алгоритмі?

10. Як визначити гарантовану мінімальну частоту елемента?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму Space-Saving для відстеження частих елементів

2. Мета

Вивчити принципи роботи детермінованих алгоритмів для потоків даних, реалізувати ефективну версію Space-Saving, дослідити точність оцінок та порівняти з іншими алгоритмами для знаходження частих елементів.

3. Завдання

Реалізувати алгоритм Space-Saving з ефективними структурами даних, створити модифікації для різних сценаріїв, провести експерименти з точністю оцінок, розробити візуалізацію роботи алгоритму.

4. Опис

Лабораторна робота включає реалізацію Space-Saving з використанням хеш-таблиць та двозв'язних списків, дослідження впливу параметра k на точність, створення розширень для практичних застосувань.

5. Порядок виконання

- Реалізувати базові структури Counter, Bucket, SpaceSaving
- Створити ефективне управління buckets
- Реалізувати операції processElement та заміни
- Додати функції запитів getTopK та getFrequentItems
- Створити генератори потоків з різними розподілами
- Реалізувати візуалізацію внутрішнього стану
- Провести експерименти з точністю для різних k
- Порівняти з Lossy Counting та Count-Min Sketch
- Реалізувати версію для sliding window
- Створити демонстраційний застосунок

6. Результат

Працююча реалізація Space-Saving з оптимізованими структурами даних, графіки точності оцінок для різних розподілів, порівняльний аналіз з

іншими алгоритмами, візуалізація роботи, рекомендації щодо вибору параметрів.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас SpaceSaving з усіма операціями
- Ефективне управління buckets
- Обробка граничних випадків
- Модифікації для sliding window
- Тестові генератори потоків

2. Продемонструвати роботу додатку:

- Відстеження top-k елементів у реальному часі
- Візуалізація зміни частот та помилок
- Порівняння з точними підрахунками
- Робота з різними типами даних
- Аналіз використання пам’яті

3. Бути готовим відповісти на питання:

- Як вибрати оптимальне k для конкретної задачі?
- Чому важливо відстежувати error для кожного елемента?
- Коли Space-Saving дає кращі результати за інші алгоритми?
- Як оптимізувати структури даних для швидкодії?

Тести

1. Що відбувається коли новий елемент додається до заповненої структури Space-Saving?

- 1) Структура розширюється
- 2) Елемент ігнорується
- 3) Замінюється елемент з мінімальною частотою
- 4) Видаляється випадковий елемент

2. Яке значення error для нового елемента при заміні?

- 1) 0
- 2) 1

- 3) Частота замінюваного елемента

- 4) n/k

3. Space-Saving гарантує збереження елементів з частотою:

- 1)

- 2)

- 3)

- 4) ⩾ n/k

4. Яка амортизована складність операції processElement?

- 1) O(log k)

- 2) O(k)

- 3) O(1)

- 4) O(log n)

5. Для чого використовуються buckets в Space-Saving?

- 1) Для сортування елементів

- 2) Для групування елементів з однаковою частотою

- 3) Для хешування

- 4) Для стиснення даних

Відповіді:

1-3,

2-3,

3-2,

4-3,

5-2

Тести підвищеної складності

1. Чому гарантована мінімальна частота = count - error?

- 1) Це визначення error

- 2) error показує максимальну переоцінку через заміни

- 3) Для спрощення обчислень

- 4) Це емпіричне правило

2. Як Space-Saving обробляє елементи з однаковими частотами при заміні?

- 1) Вибирає випадково
- 2) Вибирає перший у bucket
- 3) Вибирає останній доданий
- 4) Не має значення для коректності

3. Яка максимальна похибка для елемента в Space-Saving?
- 1) k
- 2) n
- 3) n/k
- 4) log n

4. Що відбувається з сумою всіх лічильників при обробці нового елемента?
- 1) Залишається незмінною
- 2) Збільшується на 1
- 3) Може зменшитися
- 4) Збільшується на k

5. Як ефективно знайти елемент з мінімальною частотою?
- 1) Перебрати всі елементи
- 2) Використати min-heap
- 3) Використати перший bucket у впорядкованому списку
- 4) Використати хеш-таблицю

6. Чому Space-Saving називається “space-saving”?
- 1) Використовує стиснення
- 2) Використовує фіксовану кількість лічильників
- 3) Видаляє дублікати
- 4) Використовує мало пам’яті

7. Як змінюється точність при збільшенні k?
- 1) Погіршується
- 2) Покращується
- 3) Не змінюється
- 4) Змінюється випадково

8. Що робити для підтримки точних top-k якщо k змінюється динамічно?

- 1) Перебудувати структуру
- 2) Використовувати Space-Saving з запасом
- 3) Неможливо підтримувати
- 4) Використовувати кілька структур

9. Як об’єднати два Space-Saving для розподілених систем?
- 1) Просто додати лічильники
- 2) Використати спеціальний алгоритм злиття
- 3) Неможливо об’єднати
- 4) Взяти максимум лічильників

10. Яка основна відмінність від Count-Min Sketch?
- 1) Space-Saving повільніший
- 2) Space-Saving зберігає точні елементи, а не хеші
- 3) Space-Saving використовує більше пам’яті
- 4) Space-Saving імовірнісний

Відповіді:

1. 2) error відстежує скільки разів елемент “успадкував” частоту при замінах
2. 4) Будь-який вибір зберігає коректність алгоритму
3. 3) Максимальна переоцінка обмежена n/k
4. 2) Кожен новий елемент збільшує загальну суму на 1
5. 3) Buckets впорядковані за частотою, перший містить мінімальні
6. 2) Економить простір використовуючи фіксовану кількість лічильників
7. 2) Більше k означає точніше відстеження та меншу похибку
8. 2) Використовувати Space-Saving з k=max_possible_k
9. 2) Потрібен складний алгоритм що враховує errors
10. 2) Space-Saving зберігає самі елементи, не використовує хешування

Завдання для самостійної роботи

1. Реалізуйте Space-Saving з підтримкою декременту лічильників для оновлюваних потоків.
2. Створіть паралельну версію Space-Saving з lock-free структурами даних.
3. Розробіть Space-Saving для k-грам з ефективним використанням пам’яті.

4. Реалізуйте Filtered Space-Saving що ігнорує елементи за певними критеріями.

5. Створіть адаптивну версію що автоматично налаштовує k.

6. Розробіть Space-Saving з підтримкою персистентності для відновлення після збоїв.

7. Реалізуйте ієрархічний Space-Saving для багаторівневої агрегації.

8. Створіть версію з підтримкою TTL для елементів.

9. Розробіть Space-Saving для географічно розподілених даних.

10. Дослідіть використання Space-Saving для стиснення даних.

Теми лабораторних робіт

1. Базова реалізація Space-Saving та аналіз точності.

2. Порівняння Space-Saving, Lossy Counting та Frequent алгоритмів.

3. Розробка системи кешування на основі Space-Saving.

4. Створення монітора мережевого трафіку з Space-Saving.

5. Реалізація рекомендаційної системи з відстеженням популярності.

6. Аналізатор логів з використанням Space-Saving для різних метрик.

7. Розробка системи виявлення трендів у соціальних мережах.

8. Створення оптимізатора запитів БД на основі Space-Saving.

9. Реалізація розподіленого Space-Saving для кластерних систем.

10. Дослідження Space-Saving для обробки потоків IoT даних.

14.3. Структури для зовнішньої пам’яті

Ми переходимо від сценаріїв, де дані ледь поміщаються в оперативну пам’ять або обробляються “на льоту”, до іншої фундаментальної проблеми великих даних: коли набір даних настільки великий, що зберігається на зовнішній пам’яті — жорсткому диску (HDD) або твердотільному накопичувачі (SSD).

У цьому випадку головним вузьким місцем стає не обсяг RAM, а надзвичайно повільна швидкість доступу до диска. Операція читання з диска може бути в тисячі разів повільнішою за доступ до оперативної пам’яті. Тому ефективність алгоритму вимірюється не стільки кількістю процесорних операцій, скільки кількістю операцій дискового вводу-виводу.

Цей розділ присвячений структурам та алгоритмам, розробленим спеціально для мінімізації дискових доступів. Ми розглянемо:

- Зовнішнє сортування злиттям: Класичний алгоритм, що дозволяє відсортувати гігабайтні файли, які неможливо завантажити в пам'ять цілком.

- LSM-дерева (Log-Structured Merge-trees): Сучасна структура даних, що лежить в основі багатьох високопродуктивних баз даних (наприклад, Cassandra, RocksDB). Вона геніально оптимізує операції запису, перетворюючи хаотичні точкові записи на впорядковані послідовні потоки на диску.

Розуміння цих підходів є ключовим для побудови будь-якої системи, що працює з великими обсягами постійних даних.

LSM-дерева

Теоретичні відомості

Log-Structured Merge-tree (LSM-дерево) є структурою даних, оптимізованою для систем з високою інтенсивністю запису, де дані зберігаються на повільних пристроях зовнішньої пам'яті. Розроблена Патріком О'Нілом, Едвардом Ченгом, Дітером Голлманом та Елізабет О'Ніл у 1996 році, ця структура стала основою для багатьох сучасних NoSQL баз даних та систем зберігання великих даних.

Основна ідея LSM-дерева

Традиційні B-дерева та їх варіанти оптимізовані для читання, але страждають від проблеми випадкового доступу при записі, що призводить до великої кількості операцій введення-виведення. LSM-дерево вирішує цю проблему, розділяючи структуру на кілька компонентів різного розміру, де записи спочатку накопичуються в пам'яті, а потім послідовно записуються на диск великими блоками.

Ключова ідея полягає в тому, що всі записи (вставки, оновлення, видалення) спочатку потрапляють у структуру в пам'яті, яка називається MemTable. Коли MemTable досягає певного розміру, вона записується на диск як незмінний відсортований файл, який називається SSTable (Sorted String Table). З часом ці файли об'єднуються через процес компактифікації, що дозволяє підтримувати ефективність читання.

Архітектура LSM-дерева

LSM-дерево складається з декількох рівнів, де кожен наступний рівень має більший розмір. Типова архітектура включає компонент C0 в оперативній пам’яті та послідовність компонентів C1, C2, …, Ck на диску. Розмір кожного наступного рівня зазвичай в T разів більший за попередній, де T називається фактором росту (зазвичай від 2 до 10).

```
struct LSMTree {
    memTable: MemTable           // C0 - компонент в пам'яті
    immutableMemTable: MemTable  // заморожена memTable для flush
    levels: array of Level       // рівні на диску
    config: LSMConfig           // конфігурація дерева
    manifest: Manifest          // метадані про файли
}

struct MemTable {
    data: SkipList or RedBlackTree  // впорядкована структура в пам'я
ті
    size: integer                    // поточний розмір
    minKey: Key                      // мінімальний ключ
    maxKey: Key                      // максимальний ключ
}

struct Level {
    levelNumber: integer
    sstables: array of SSTable
    totalSize: integer
    maxSize: integer
}

struct SSTable {
    fileName: string
    minKey: Key
    maxKey: Key
    bloomFilter: BloomFilter
    index: SparseIndex
    dataBlocks: array of DataBlock
}

struct LSMConfig {
    memTableSize: integer      // максимальний розмір MemTable
    level0FileCount: integer   // кількість файлів для тригеру компак
тифікації L0
    levelSizeMultiplier: integer  // фактор росту між рівнями
    maxLevels: integer         // максимальна кількість рівнів
```

```
    blockSize: integer          // розмір блоку даних
}
```

Операція PUT (запис)

Операція запису в LSM-дерево дуже ефективна, оскільки вона просто додає запис до структури в пам'яті. Важливо, що LSM-дерево не модифікує існуючі дані на диску - замість цього воно записує нову версію даних, що робить структуру append-only.

```
function put(lsm: LSMTree, key: Key, value: Value)
    // Створюємо запис з timestamp для версіонування
    record = Record(
        key = key,
        value = value,
        timestamp = getCurrentTimestamp(),
        type = PUT
    )

    // Перевіряємо чи потрібен flush
    if lsm.memTable.size >= lsm.config.memTableSize then
        // Заморожуємо поточну MemTable
        lsm.immutableMemTable = lsm.memTable
        lsm.memTable = createNewMemTable()

        // Запускаємо фоновий flush
        scheduleBackgroundFlush(lsm)

    // Додаємо запис до MemTable
    lsm.memTable.data.insert(record)
    lsm.memTable.size += sizeOf(record)

    // Оновлюємо межі ключів
    updateKeyBounds(lsm.memTable, key)

    // Записуємо в Write-Ahead Log для відновлення
    appendToWAL(record)
```

Операція GET (читання)

Читання з LSM-дерева складніше, оскільки потрібно перевірити кілька місць: спочатку MemTable, потім незмінну MemTable (якщо є), і нарешті SSTable на кожному рівні, починаючи з найновішого. Для оптимізації використовуються фільтри Блума та індекси.

```
function get(lsm: LSMTree, key: Key): Value or null
    // Перевіряємо MemTable
    result = lsm.memTable.data.find(key)
```

```
    if result != null then
        if result.type == DELETE then
            return null  // ключ був видалений
        return result.value

    // Перевіряємо immutable MemTable якщо існує
    if lsm.immutableMemTable != null then
        result = lsm.immutableMemTable.data.find(key)
        if result != null then
            if result.type == DELETE then
                return null
            return result.value

    // Пошук на диску по рівнях
    for level in lsm.levels do
        // Сортуємо SSTable за timestamp (новіші спочатку)
        sortedSSTables = sortByTimestampDesc(level.sstables)

        for sstable in sortedSSTables do
            // Швидка перевірка діапазону ключів
            if key < sstable.minKey or key > sstable.maxKey then
                continue

            // Перевірка Bloom фільтра
            if not sstable.bloomFilter.mightContain(key) then
                continue

            // Пошук в SSTable
            value = searchInSSTable(sstable, key)
            if value != null then
                if value.type == DELETE then
                    return null
                return value.value

    return null  // ключ не знайдено

function searchInSSTable(sstable: SSTable, key: Key): Record or null
    // Використовуємо розріджений індекс для знаходження блоку
    blockIndex = sstable.index.findBlockForKey(key)
    if blockIndex == -1 then
        return null

    // Завантажуємо блок даних
    block = loadDataBlock(sstable.fileName, blockIndex)
```

```
    // Бінарний пошук в блоці
    return binarySearch(block, key)
```

Flush процес

Коли MemTable досягає максимального розміру, вона перетворюється на SSTable і записується на диск. Цей процес називається flush і виконується у фоновому режимі.

```
function flushMemTable(lsm: LSMTree)
    if lsm.immutableMemTable == null then
        return

    // Створюємо новий SSTable
    sstableBuilder = SSTableBuilder(lsm.config.blockSize)

    // Ітеруємо по всіх записах у відсортованому порядку
    iterator = lsm.immutableMemTable.data.iterator()
    while iterator.hasNext() do
        record = iterator.next()

        // Додаємо до Bloom фільтра
        sstableBuilder.bloomFilter.add(record.key)

        // Додаємо запис
        sstableBuilder.addRecord(record)

        // Якщо блок заповнений, записуємо його
        if sstableBuilder.currentBlockSize >= lsm.config.blockSize th
en
            sstableBuilder.flushBlock()

    // Записуємо останній блок
    sstableBuilder.flushBlock()

    // Фіналізуємо SSTable
    sstable = sstableBuilder.finish()

    // Додаємо до рівня 0
    lsm.levels[0].sstables.append(sstable)
    lsm.levels[0].totalSize += sstable.size

    // Очищаємо immutable MemTable
    lsm.immutableMemTable = null

    // Перевіряємо чи потрібна компактифікація
```

```
    if needsCompaction(lsm.levels[0]) then
        scheduleCompaction(lsm, 0)
```

Компактифікація

Компактифікація є критичним процесом для підтримки ефективності LSM-дерева. Вона об'єднує кілька SSTable, видаляє застарілі версії та видалені записи, створюючи нові оптимізовані SSTable.

```
function compactLevel(lsm: LSMTree, level: integer)
    currentLevel = lsm.levels[level]
    nextLevel = lsm.levels[level + 1]

    // Вибираємо файли для компактифікації
    if level == 0 then
        // Для L0 беремо всі файли що перекриваються
        filesToCompact = selectAllOverlappingFiles(currentLevel)
    else
        // Для інших рівнів використовуємо round-robin або size-based
        filesToCompact = selectFilesForCompaction(currentLevel)

    // Знаходимо діапазон ключів
    keyRange = getKeyRange(filesToCompact)

    // Знаходимо перекриваючі файли на наступному рівні
    overlappingFiles = findOverlappingFiles(nextLevel, keyRange)

    // Створюємо merge iterator
    iterator = MergeIterator()
    for file in filesToCompact do
        iterator.addIterator(file.iterator())
    for file in overlappingFiles do
        iterator.addIterator(file.iterator())

    // Виконуємо merge
    newSSTables = []
    builder = SSTableBuilder(lsm.config.blockSize)

    lastKey = null
    while iterator.hasNext() do
        record = iterator.next()

        // Пропускаємо застарілі версії та видалені записи
        if lastKey != null and lastKey == record.key then
            continue  // старіша версія
```

```
        if record.type == DELETE and isOldEnough(record) then
            continue  // можна видалити tombstone

        builder.addRecord(record)
        lastKey = record.key

        if builder.size >= getTargetFileSize(level + 1) then
            newSSTables.append(builder.finish())
            builder = SSTableBuilder(lsm.config.blockSize)

    if builder.hasRecords() then
        newSSTables.append(builder.finish())

    // Атомарно оновлюємо manifest
    updateManifest(lsm, {
        deleteFiles: filesToCompact + overlappingFiles,
        addFiles: newSSTables,
        level: level + 1
    })

    // Видаляємо старі файли
    for file in filesToCompact + overlappingFiles do
        deleteFile(file)
```

Стратегії компактифікації

Існує кілька стратегій компактифікації, кожна з яких має свої переваги:

1. Leveled Compaction

Кожен рівень має фіксований розмір, і файли не перекриваються за ключами в межах рівня (крім L0).

```
function leveledCompaction(lsm: LSMTree, level: integer)
    while lsm.levels[level].totalSize > lsm.levels[level].maxSize do
        // Вибираємо файл з найменшим ключем
        file = selectSmallestKeyFile(lsm.levels[level])

        // Компактуємо з наступним рівнем
        compactFileWithLevel(lsm, file, level, level + 1)
```

2. Size-Tiered Compaction

Файли групуються за розміром, і коли накопичується достатньо файлів схожого розміру, вони об’єднуються.

```
function sizeTieredCompaction(lsm: LSMTree, level: integer)
    // Групуємо файли за розміром
    buckets = groupFilesBySize(lsm.levels[level].sstables)
```

```
    for bucket in buckets do
        if bucket.fileCount >= lsm.config.minFilesForCompaction then
            mergeFiles(bucket.files)
```

Оптимізації

1. Bloom фільтри

Bloom фільтри дозволяють швидко визначити, чи може ключ бути в SSTable, без читання файлу.

```
struct BloomFilter {
    bitArray: array of boolean
    hashFunctions: array of HashFunction
    size: integer
    numHashFunctions: integer
}

function createBloomFilter(expectedElements: integer, falsePositiveRa
te: float): BloomFilter
    // Оптимальний розмір
    size = -expectedElements * ln(falsePositiveRate) / (ln(2)^2)

    // Оптимальна кількість хеш-функцій
    numHashFunctions = (size / expectedElements) * ln(2)

    return BloomFilter(
        bitArray = array[size],
        hashFunctions = generateHashFunctions(numHashFunctions),
        size = size,
        numHashFunctions = numHashFunctions
    )
```

2. Block кешування

Часто використовувані блоки даних кешуються в пам’яті для прискорення читання.

```
struct BlockCache {
    cache: LRUCache<BlockId, DataBlock>
    maxSize: integer
}

function readBlock(lsm: LSMTree, sstable: SSTable, blockIndex: intege
r): DataBlock
    blockId = BlockId(sstable.fileName, blockIndex)

    // Перевіряємо кеш
```

```
    block = lsm.blockCache.get(blockId)
    if block != null then
        return block

    // Читаємо з диску
    block = loadDataBlock(sstable.fileName, blockIndex)

    // Додаємо до кешу
    lsm.blockCache.put(blockId, block)

    return block
```

Область застосування

LSM-дерева широко використовуються в сучасних системах зберігання даних:

- NoSQL бази даних: Apache Cassandra, HBase, LevelDB, RocksDB використовують LSM-дерева як основну структуру зберігання.

- Системи обробки потоків: Apache Kafka використовує LSM-подібні структури для зберігання логів.

- Часові ряди: InfluxDB та інші бази даних часових рядів використовують LSM для ефективного запису даних з часовими мітками.

- Пошукові системи: Elasticsearch та Lucene використовують LSM-подібні підходи для індексації.

- Блокчейн: Багато блокчейн-платформ використовують LSM-дерева для зберігання стану.

- Аналітичні системи: Системи OLAP використовують LSM для ефективного завантаження даних.

Приклади застосування

Приклад 1: Базова робота LSM-дерева

Розглянемо послідовність операцій з простим LSM-деревом:

```
Конфігурація:
- MemTable size: 4 записи
- Level size multiplier: 2
- Max levels: 3

Операції:
1. put(A, 1)
   MemTable: [(A,1)]
```

```
2. put(B, 2)
   MemTable: [(A,1), (B,2)]

3. put(C, 3)
   MemTable: [(A,1), (B,2), (C,3)]

4. put(D, 4)
   MemTable: [(A,1), (B,2), (C,3), (D,4)]

5. put(E, 5) - тригер flush
   MemTable заморожується
   Створюється SSTable_1: [(A,1), (B,2), (C,3), (D,4)]
   Level 0: [SSTable_1]
   MemTable: [(E,5)]

6. put(A, 10) - оновлення
   MemTable: [(A,10), (E,5)]

7. get(A)
   - Перевірка MemTable: знайдено (A,10)
   - Повертаємо 10

8. put(F, 6), put(G, 7)
   MemTable: [(A,10), (E,5), (F,6), (G,7)]

9. put(H, 8) - тригер flush
   Створюється SSTable_2: [(A,10), (E,5), (F,6), (G,7)]
   Level 0: [SSTable_1, SSTable_2]
   MemTable: [(H,8)]

10. Компактифікація L0->L1
    Merge SSTable_1 + SSTable_2:
    - (A,1) vs (A,10) -> залишаємо (A,10)
    - Результат: [(A,10), (B,2), (C,3), (D,4), (E,5), (F,6), (G,7)]
    Level 1: [SSTable_3]
    Level 0: []
```

Приклад 2: Key-Value сховище

Реалізація простого key-value сховища на основі LSM-дерева:

```
struct KeyValueStore {
    lsm: LSMTree
    wal: WriteAheadLog
    config: StoreConfig
}
```

```
function initKeyValueStore(dataDir: string, config: StoreConfig): Key
ValueStore
    // Створюємо директорії
    createDirectories(dataDir)

    // Ініціалізуємо WAL
    wal = initWAL(dataDir + "/wal")

    // Ініціалізуємо LSM
    lsmConfig = LSMConfig(
        memTableSize = config.memTableSizeMB * 1024 * 1024,
        level0FileCount = 4,
        levelSizeMultiplier = 10,
        maxLevels = 7,
        blockSize = 4096
    )

    lsm = initLSMTree(dataDir, lsmConfig)

    // Відновлюємо з WAL якщо потрібно
    recoverFromWAL(lsm, wal)

    return KeyValueStore(lsm, wal, config)

function putBatch(store: KeyValueStore, batch: array of (Key, Value))
    // Записуємо batch в WAL
    walRecord = createBatchRecord(batch)
    store.wal.append(walRecord)

    // Застосовуємо до LSM
    for (key, value) in batch do
        put(store.lsm, key, value)

    // Синхронізуємо WAL
    store.wal.sync()

function delete(store: KeyValueStore, key: Key)
    // Записуємо tombstone
    record = Record(
        key = key,
        value = null,
        timestamp = getCurrentTimestamp(),
        type = DELETE
    )
```

```
    store.wal.append(record)
    store.lsm.memTable.insert(record)

function scan(store: KeyValueStore, startKey: Key, endKey: Key): Iter
ator
    // Створюємо merge iterator для всіх компонентів
    iterators = []

    // MemTable iterator
    iterators.append(store.lsm.memTable.data.rangeIterator(startKey,
endKey))

    // Immutable MemTable iterator
    if store.lsm.immutableMemTable != null then
        iterators.append(store.lsm.immutableMemTable.data.rangeIterat
or(startKey, endKey))

    // SSTable iterators
    for level in store.lsm.levels do
        for sstable in level.sstables do
            if overlapsRange(sstable, startKey, endKey) then
                iterators.append(createSSTableIterator(sstable, start
Key, endKey))

    return MergeIterator(iterators)
```

Приклад 3: Часові ряди

LSM-дерево для зберігання даних часових рядів:

```
struct TimeSeriesLSM {
    lsm: LSMTree
    retentionPolicy: RetentionPolicy
    downsampling: DownsamplingConfig
}

function writeTimeSeriesPoint(ts: TimeSeriesLSM, metric: string, time
stamp: integer, value: float, tags: map)
    // Створюємо композитний ключ
    key = createTimeSeriesKey(metric, tags, timestamp)

    // Кодуємо значення
    encodedValue = encodeTimeSeriesValue(value, tags)

    // Записуємо
    put(ts.lsm, key, encodedValue)
```

```
    // Перевіряємо retention policy
    if shouldCheckRetention(ts) then
        scheduleRetentionCleanup(ts)

function createTimeSeriesKey(metric: string, tags: map, timestamp: in
teger): Key
    // Формат: metric|tag1=value1,tag2=value2|timestamp
    tagString = serializeTags(tags)
    return metric + "|" + tagString + "|" + timestamp

function queryTimeSeries(ts: TimeSeriesLSM, metric: string, tags: map
,
                         startTime: integer, endTime: integer): array
of Point
    // Створюємо діапазон ключів
    startKey = createTimeSeriesKey(metric, tags, startTime)
    endKey = createTimeSeriesKey(metric, tags, endTime)

    // Скануємо діапазон
    iterator = scan(ts.lsm, startKey, endKey)
    points = []

    while iterator.hasNext() do
        entry = iterator.next()

        // Декодуємо
        point = decodeTimeSeriesEntry(entry)

        // Фільтруємо за тегами
        if matchesTags(point.tags, tags) then
            points.append(point)

    return points

function downsampleTimeSeries(ts: TimeSeriesLSM, metric: string,
                              interval: integer, aggregation: Aggregat
ionType)
    // Спеціальна компактифікація для downsampling
    iterator = scanMetric(ts.lsm, metric)

    currentBucket = null
    bucketValues = []

    while iterator.hasNext() do
        point = iterator.next()
```

```
        bucket = point.timestamp / interval

        if bucket != currentBucket then
            if currentBucket != null then
                // Агрегуємо попередній bucket
                aggregatedValue = aggregate(bucketValues, aggregation
)
                downsampledKey = createDownsampledKey(metric, current
Bucket, interval)
                put(ts.lsm, downsampledKey, aggregatedValue)

            currentBucket = bucket
            bucketValues = []

        bucketValues.append(point.value)
```

Приклад 4: Вторинні індекси

Реалізація вторинних індексів поверх LSM-дерева:

```
struct IndexedLSM {
    primaryLSM: LSMTree      // основні дані
    indexes: map<string, LSMTree>  // вторинні індекси
}

function createIndex(ilsm: IndexedLSM, indexName: string,
                     keyExtractor: function(Value) -> IndexKey)
    // Створюємо новий LSM для індексу
    indexLSM = initLSMTree(getIndexPath(indexName))
    ilsm.indexes[indexName] = indexLSM

    // Переіндексуємо існуючі дані
    iterator = fullScan(ilsm.primaryLSM)

    while iterator.hasNext() do
        entry = iterator.next()
        indexKey = keyExtractor(entry.value)

        // Індекс зберігає indexKey -> primaryKey
        put(indexLSM, indexKey, entry.key)

function putWithIndexes(ilsm: IndexedLSM, key: Key, value: Value)
    // Отримуємо старе значення для оновлення індексів
    oldValue = get(ilsm.primaryLSM, key)

    // Записуємо в основний LSM
    put(ilsm.primaryLSM, key, value)
```

```
    // Оновлюємо індекси
    for (indexName, indexLSM) in ilsm.indexes do
        keyExtractor = getKeyExtractor(indexName)

        // Видаляємо старий індексний запис
        if oldValue != null then
            oldIndexKey = keyExtractor(oldValue)
            delete(indexLSM, oldIndexKey)

        // Додаємо новий індексний запис
        newIndexKey = keyExtractor(value)
        put(indexLSM, newIndexKey, key)

function queryByIndex(ilsm: IndexedLSM, indexName: string,
                      indexKey: IndexKey): array of (Key, Value)
    indexLSM = ilsm.indexes[indexName]

    // Знаходимо primary keys через індекс
    primaryKeys = []
    iterator = scan(indexLSM, indexKey, incrementKey(indexKey))

    while iterator.hasNext() do
        entry = iterator.next()
        if entry.key != indexKey then
            break
        primaryKeys.append(entry.value)

    // Отримуємо значення з основного LSM
    results = []
    for primaryKey in primaryKeys do
        value = get(ilsm.primaryLSM, primaryKey)
        if value != null then
            results.append((primaryKey, value))

    return results
```

Переваги та недоліки

Переваги

- Висока швидкість запису: Послідовний запис на диск набагато швидший за випадковий доступ.

- Ефективне використання простору: Компактифікація видаляє застарілі версії та оптимізує розміщення даних.

- Природна підтримка версіонування: Кожен запис має timestamp, що дозволяє легко реалізувати MVCC.

- Стійкість до збоїв: Write-ahead logging забезпечує durability та можливість відновлення.

- Гнучкість налаштування: Різні стратегії компактифікації дозволяють оптимізувати під конкретні use cases.

- Ефективність для SSD: Знижує write amplification порівняно з B-деревами.

Недоліки

- Повільніше читання: Потрібно перевірити кілька рівнів для знаходження даних.

- Write amplification: Дані переписуються кілька разів під час компактифікації.

- Space amplification: Тимчасово може використовуватися більше простору через дублювання під час компактифікації.

- Складність налаштування: Багато параметрів впливають на продуктивність.

- Затримки через компактифікацію: Фонові процеси можуть впливати на латентність операцій.

- Неефективність для малих датасетів: Overhead структури може бути значним для невеликих обсягів даних.

Контрольні запитання

1. Що таке LSM-дерево та яка його основна перевага над B-деревами?

2. Поясніть роль MemTable та SSTable в архітектурі LSM-дерева.

3. Як працює процес flush та чому він важливий?

4. Що таке компактифікація та які існують стратегії її виконання?

5. Як LSM-дерево забезпечує ефективність операцій запису?

6. Чому читання з LSM-дерева повільніше ніж запис?

7. Яку роль відіграють Bloom фільтри в оптимізації читання?

8. Що таке write amplification та як його мінімізувати?

9. Як LSM-дерево підтримує транзакційність та MVCC?

10. Які фактори впливають на вибір між leveled та size-tiered компактифікацією?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження LSM-дерева для систем зберігання даних

2. Мета

Вивчити принципи роботи LSM-дерев, реалізувати базову версію з підтримкою основних операцій, дослідити вплив різних стратегій компактифікації на продуктивність.

3. Завдання

Реалізувати LSM-дерево з MemTable, SSTable, процесами flush та компактифікації. Створити key-value сховище на основі LSM-дерева. Провести експерименти з різними workloads та налаштуваннями.

4. Опис

Лабораторна робота включає поетапну реалізацію компонентів LSM-дерева, починаючи з простих структур даних і закінчуючи повноцінним сховищем з підтримкою персистентності та оптимізацій.

5. Порядок виконання

- Реалізувати MemTable на основі skip list або червоно-чорного дерева
- Створити формат SSTable з блоками даних та індексом
- Реалізувати процес flush з MemTable в SSTable
- Додати Write-Ahead Log для відновлення
- Реалізувати операції put, get, delete
- Створити просту leveled компактифікацію
- Додати Bloom фільтри для оптимізації
- Реалізувати block cache
- Провести benchmarks для різних операцій
- Порівняти з B-tree реалізацією

6. Результат

Працююче LSM-дерево з візуалізацією структури рівнів, графіки продуктивності для різних операцій, аналіз write/read/space amplification, рекомендації щодо налаштування для різних workloads.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Реалізація MemTable з ефективною структурою даних
- SSTable формат з блоками та індексацією
- Процеси flush та компактифікації
- Write-Ahead Log для durability
- Оптимізації (Bloom filter, cache)

2. Продемонструвати роботу додатку:
- CRUD операції з key-value сховищем
- Візуалізація процесу flush та компактифікації
- Моніторинг метрик (write/read amplification)
- Відновлення після збою
- Порівняння продуктивності з B-tree

3. Бути готовим відповісти на питання:
- Як вибрати оптимальний розмір MemTable?
- Коли тригерувати компактифікацію?
- Як балансувати між read та write performance?
- Які компроміси різних стратегій компактифікації?

Тести

1. Куди спочатку потрапляють нові записи в LSM-дереві?
- 1) Безпосередньо на диск
- 2) В MemTable в пам’яті
- 3) В SSTable
- 4) В індекс

2. Що відбувається коли MemTable досягає максимального розміру?
- 1) Видаляються старі записи
- 2) Блокуються нові записи
- 3) MemTable flush на диск як SSTable
- 4) Збільшується розмір MemTable

3. Яка основна мета компактифікації?
- 1) Стиснення даних
- 2) Видалення застарілих версій та реорганізація

- 3) Прискорення запису
- 4) Створення резервних копій

4. Що таке tombstone в контексті LSM-дерева?
- 1) Пошкоджений запис
- 2) Маркер видалення
- 3) Тимчасовий файл
- 4) Індексний запис

5. Яка структура даних зазвичай використовується для MemTable?
- 1) Hash table
- 2) B-tree
- 3) Skip list або Red-Black tree
- 4) Array

Відповіді:

1-2,

2-3,

3-2,

4-2,

5-3

Тести підвищеної складності

1. Чому LSM-дерева мають write amplification?
- 1) Через шифрування даних
- 2) Дані переписуються кілька разів під час компактифікації
- 3) Через дублювання для надійності
- 4) Через індексацію

2. Що таке leveled compaction strategy?
- 1) Компактифікація за розміром файлів
- 2) Кожен рівень має фіксований розмір, файли не перекриваються
- 3) Випадкова компактифікація
- 4) Компактифікація за часом

3. Як Bloom filter допомагає в LSM-дереві?

- 1) Стискає дані
- 2) Швидко визначає відсутність ключа без читання SSTable
- 3) Прискорює запис
- 4) Шифрує дані

4. Чому читання з LSM повільніше за B-tree?
- 1) Використовується повільніша структура даних
- 2) Потрібно перевірити кілька рівнів SSTable
- 3) Дані зашифровані
- 4) Немає індексів

5. Що таке Write-Ahead Log?
- 1) Кеш для читання
- 2) Журнал змін для відновлення після збоїв
- 3) Індекс для пошуку
- 4) Тимчасовий буфер

6. Як LSM-дерево підтримує MVCC?
- 1) Використовує блокування
- 2) Зберігає кілька версій з timestamps
- 3) Копіює всю базу даних
- 4) Не підтримує

7. Що означає space amplification в LSM?
- 1) Стиснення даних
- 2) Тимчасове використання додаткового простору
- 3) Фрагментація диску
- 4) Збільшення RAM

8. Коли можна видалити tombstone?
- 1) Одразу після створення
- 2) Коли він досяг найнижчого рівня і старший за GC період
- 3) Ніколи
- 4) При кожній компактифікації

9. Яка перевага size-tiered compaction?

- 1) Менше write amplification
- 2) Краще для читання
- 3) Простіша реалізація
- 4) Менше використання диску

10. Як оптимізувати LSM для SSD?
- 1) Збільшити розмір блоків
- 2) Зменшити write amplification
- 3) Відключити компактифікацію
- 4) Використовувати більше рівнів

Відповіді:

1. 2) Кожен запис переписується при переміщенні між рівнями
2. 2) Рівні мають обмеження розміру, ключі не перекриваються в рівні
3. 2) Дозволяє уникнути читання SSTable якщо ключа там немає
4. 2) На відміну від B-tree, потрібно перевірити кілька файлів
5. 2) WAL забезпечує durability та recovery
6. 2) Кожен запис має версію/timestamp
7. 2) Під час компактифікації тимчасово існують старі та нові файли
8. 2) Tombstone можна видалити коли гарантовано немає старіших версій
9. 1) Менше переписування даних порівняно з leveled
10. 2) SSD мають обмежену кількість циклів запису

Завдання для самостійної роботи

1. Реалізуйте Universal Compaction стратегію та порівняйте з Leveled.
2. Створіть адаптивну компактифікацію що перемикається між стратегіями.
3. Розробіть LSM-дерево оптимізоване для time-series workload.
4. Реалізуйте distributed LSM з sharding та replication.
5. Створіть in-memory LSM для кешування з persistence.
6. Розробіть column-family підтримку як в RocksDB.
7. Реалізуйте merge operator для атомарних read-modify-write.
8. Створіть backup/restore функціональність з incremental backups.
9. Розробіть monitoring dashboard для LSM метрик.

10. Дослідіть використання машинного навчання для оптимізації компактифікації.

Теми лабораторних робіт

1. Базова реалізація LSM-дерева з візуалізацією структури.
2. Порівняльний аналіз стратегій компактифікації.
3. Розробка key-value сховища з підтримкою транзакцій.
4. Створення time-series database на основі LSM.
5. Реалізація вторинних індексів для LSM-дерева.
6. Оптимізація LSM для різних типів накопичувачів (HDD/SSD/NVMe).
7. Розробка розподіленого LSM-сховища.
8. Інтеграція LSM в систему обробки потоків даних.
9. Створення embedded database на основі LSM.
10. Дослідження гібридних структур LSM+B-tree.

Зовнішнє сортування злиттям

Теоретичні відомості

Зовнішнє сортування злиттям є фундаментальним алгоритмом для впорядкування великих обсягів даних, які не поміщаються в оперативну пам’ять комп’ютера. Цей алгоритм був розроблений Джоном фон Нейманом у 1945 році і залишається одним з найефективніших методів сортування даних на зовнішніх носіях інформації.

Основні концепції та проблематика

Коли обсяг даних перевищує доступну оперативну пам’ять, традиційні алгоритми внутрішнього сортування стають неефективними або взагалі неможливими для застосування. Основна проблема полягає в тому, що доступ до даних на зовнішніх носіях (жорсткі диски, SSD) значно повільніший за доступ до оперативної пам’яті. Типова швидкість доступу до оперативної пам’яті вимірюється в наносекундах, тоді як доступ до жорсткого диска займає мілісекунди, що повільніше в мільйони разів.

Зовнішнє сортування злиттям вирішує цю проблему шляхом мінімізації кількості операцій введення-виведення та максимального використання послідовного доступу до даних, який значно ефективніший за випадковий доступ на зовнішніх носіях.

Модель обчислень для зовнішньої пам’яті

Для аналізу алгоритмів зовнішньої пам’яті використовується модель, яка враховує ієрархічну структуру пам’яті комп’ютера:

- M - розмір доступної оперативної пам’яті (в елементах)

- B - розмір блоку для операцій введення-виведення (в елементах)

- N - загальна кількість елементів для сортування

- $n = N/B$ - кількість блоків даних

Ефективність алгоритму вимірюється кількістю операцій введення-виведення (I/O операцій), оскільки саме вони є вузьким місцем продуктивності.

Двофазний алгоритм зовнішнього сортування

Класичний алгоритм зовнішнього сортування злиттям складається з двох основних фаз:

Фаза 1 - Створення початкових відсортованих фрагментів (runs)

На цій фазі вхідний файл розбивається на фрагменти розміром M (розмір доступної пам’яті), кожен фрагмент завантажується в пам’ять, сортується за допомогою ефективного алгоритму внутрішнього сортування (наприклад, quicksort або heapsort), і записується назад на диск як окремий відсортований файл.

Фаза 2 - Багатошляхове злиття

На цій фазі відсортовані фрагменти об’єднуються в один великий відсортований файл. Процес злиття може виконуватися за кілька проходів, якщо кількість початкових фрагментів перевищує кількість потоків, які можна одночасно зливати.

Розглянемо детальну реалізацію кожної фази алгоритму.

```
struct ExternalSortConfig {
    memorySize: integer         // M - розмір доступної пам'яті в еле
ментах
    blockSize: integer          // B - розмір блоку I/O
    inputFile: string           // шлях до вхідного файлу
    outputFile: string          // шлях до вихідного файлу
    tempDirectory: string       // директорія для тимчасових файлів
    comparator: function        // функція порівняння елементів
}

struct Run {
```

```
    fileName: string            // ім'я файлу з відсортованим фрагмен
том
    size: integer              // кількість елементів у фрагменті
    currentPosition: integer    // поточна позиція читання
}
```

Фаза створення початкових фрагментів

Процес створення початкових відсортованих фрагментів включає читання порцій даних розміром до *M* елементів, їх сортування в пам’яті та запис на диск.

```
function createInitialRuns(config: ExternalSortConfig): array of Run
    runs = []
    runNumber = 0
    inputStream = openInputStream(config.inputFile)

    while not inputStream.isEOF() do
        // Читаємо до M елементів в пам'ять
        buffer = []
        elementsRead = 0

        while elementsRead < config.memorySize and not inputStream.is
EOF() do
            block = readBlock(inputStream, config.blockSize)
            buffer.extend(block)
            elementsRead += block.size()

        if buffer.isEmpty() then
            break

        // Сортуємо в пам'яті
        inMemorySort(buffer, config.comparator)

        // Записуємо відсортований фрагмент на диск
        runFileName = config.tempDirectory + "/run_" + runNumber
        writeRun(runFileName, buffer, config.blockSize)

        // Створюємо об'єкт Run
        run = Run(
            fileName = runFileName,
            size = buffer.size(),
            currentPosition = 0
        )
        runs.append(run)
        runNumber += 1
```

```
    closeInputStream(inputStream)
    return runs

function inMemorySort(buffer: array of elements, comparator: function
)
    // Використовуємо швидкий алгоритм внутрішнього сортування
    // Наприклад, Quicksort або Heapsort
    quickSort(buffer, 0, buffer.size() - 1, comparator)

function writeRun(fileName: string, buffer: array of elements, blockS
ize: integer)
    outputStream = openOutputStream(fileName)

    for i = 0 to buffer.size() - 1 step blockSize do
        endIndex = min(i + blockSize, buffer.size())
        block = buffer[i:endIndex]
        writeBlock(outputStream, block)

    closeOutputStream(outputStream)
```

Фаза багатошляхового злиття

Фаза злиття є найскладнішою частиною алгоритму. Вона використовує техніку k-шляхового злиття, де k - це кількість фрагментів, які можна одночасно зливати, обмежена доступною пам’яттю.

```
function multiWayMerge(runs: array of Run, config: ExternalSortConfig
): string
    if runs.isEmpty() then
        return config.outputFile

    // Обчислюємо максимальну кількість потоків для злиття
    // Резервуємо один блок для вихідного буфера
    maxStreams = (config.memorySize / config.blockSize) - 1

    currentRuns = runs
    passNumber = 0

    while currentRuns.size() > 1 do
        nextRuns = []

        // Обробляємо групи фрагментів
        for i = 0 to currentRuns.size() - 1 step maxStreams do
            endIndex = min(i + maxStreams, currentRuns.size())
            runsToMerge = currentRuns[i:endIndex]
```

```
            // Виконуємо k-шляхове злиття
            mergedRun = kWayMerge(runsToMerge, config, passNumber, i
/ maxStreams)
            nextRuns.append(mergedRun)

        // Видаляємо тимчасові файли попереднього проходу
        if passNumber > 0 then
            for run in currentRuns do
                deleteFile(run.fileName)

        currentRuns = nextRuns
        passNumber += 1

    // Перейменовуємо фінальний файл
    finalRun = currentRuns[0]
    renameFile(finalRun.fileName, config.outputFile)

    return config.outputFile
```

К-шляхове злиття

К-шляхове злиття є ключовою операцією, яка ефективно об’єднує k відсортованих потоків в один. Для цього використовується структура даних “купа” (heap) або пріоритетна черга.

```
function kWayMerge(runs: array of Run, config: ExternalSortConfig,
                   passNumber: integer, groupNumber: integer): Run
    // Створюємо вихідний файл для злитого фрагмента
    outputFileName = config.tempDirectory + "/merged_" + passNumber +
"_" + groupNumber
    outputStream = openOutputStream(outputFileName)
    outputBuffer = []

    // Ініціалізуємо вхідні потоки та буфери
    inputStreams = []
    inputBuffers = []

    for run in runs do
        stream = openInputStream(run.fileName)
        buffer = BufferedReader(stream, config.blockSize)
        inputStreams.append(stream)
        inputBuffers.append(buffer)

    // Створюємо мін-купу для ефективного вибору мінімального елемент
a
```

```
heap = MinHeap(config.comparator)

// Ініціалізуємо купу першими елементами з кожного потоку
for i = 0 to runs.size() - 1 do
    if inputBuffers[i].hasNext() then
        element = inputBuffers[i].next()
        heap.insert(HeapNode(element, i))

totalElements = 0

// Основний цикл злиття
while not heap.isEmpty() do
    // Витягуємо мінімальний елемент
    minNode = heap.extractMin()

    // Додаємо до вихідного буфера
    outputBuffer.append(minNode.element)
    totalElements += 1

    // Якщо буфер заповнений, записуємо на диск
    if outputBuffer.size() >= config.blockSize then
        writeBlock(outputStream, outputBuffer)
        outputBuffer.clear()

    // Читаємо наступний елемент з того ж потоку
    streamIndex = minNode.streamIndex
    if inputBuffers[streamIndex].hasNext() then
        nextElement = inputBuffers[streamIndex].next()
        heap.insert(HeapNode(nextElement, streamIndex))

// Записуємо залишок буфера
if not outputBuffer.isEmpty() then
    writeBlock(outputStream, outputBuffer)

// Закриваємо всі потоки
closeOutputStream(outputStream)
for stream in inputStreams do
    closeInputStream(stream)

return Run(
    fileName = outputFileName,
    size = totalElements,
    currentPosition = 0
)
```

```
struct HeapNode {
    element: any                // елемент даних
    streamIndex: integer        // індекс потоку, з якого прийшов елем
ент
}

struct BufferedReader {
    stream: InputStream
    buffer: array of elements
    bufferSize: integer
    currentIndex: integer

    function hasNext(): boolean
        if currentIndex < buffer.size() then
            return true

        // Спробувати прочитати наступний блок
        return refillBuffer()

    function next(): element
        if currentIndex >= buffer.size() then
            refillBuffer()

        element = buffer[currentIndex]
        currentIndex += 1
        return element

    function refillBuffer(): boolean
        buffer = readBlock(stream, bufferSize)
        currentIndex = 0
        return not buffer.isEmpty()
}
```

Оптимізації алгоритму

1. Заміщувальне сортування (Replacement Selection)

Заміщувальне сортування дозволяє створювати початкові фрагменти розміром більшим за доступну пам’ять, використовуючи купу.

```
function replacementSelection(config: ExternalSortConfig): array of R
un
    runs = []
    heap = MinHeap(config.comparator)
    inputStream = openInputStream(config.inputFile)
    currentRunFile = null
    currentRunSize = 0
```

```
    runNumber = 0

    // Заповнюємо купу початковими елементами
    while heap.size() < config.memorySize and not inputStream.isEOF()
do
        element = readElement(inputStream)
        heap.insert(element)

    while not heap.isEmpty() do
        // Витягуємо мінімальний елемент
        minElement = heap.extractMin()

        // Записуємо в поточний run
        if currentRunFile == null then
            currentRunFile = createNewRunFile(config, runNumber)
            runNumber += 1

        writeElement(currentRunFile, minElement)
        currentRunSize += 1

        // Читаємо новий елемент якщо можливо
        if not inputStream.isEOF() then
            newElement = readElement(inputStream)

            // Якщо новий елемент >= останнього записаного, додаємо в
купу
            if newElement >= minElement then
                heap.insert(newElement)
            else
                // Інакше відкладаємо для наступного run
                if heap.isEmpty() then
                    // Завершуємо поточний run
                    closeRunFile(currentRunFile)
                    runs.append(Run(currentRunFile.name, currentRunSi
ze, 0))

                    // Починаємо новий run
                    currentRunFile = createNewRunFile(config, runNumb
er)
                    runNumber += 1
                    currentRunSize = 0

                    heap.insert(newElement)

    // Завершуємо останній run
```

```
    if currentRunFile != null then
        closeRunFile(currentRunFile)
        runs.append(Run(currentRunFile.name, currentRunSize, 0))

    return runs
```

2. Полі-фазне сортування

Полі-фазне сортування мінімізує кількість проходів злиття, розподіляючи початкові фрагменти між кількома стрічками (файлами) за числами Фібоначчі.

```
function polyphaseMerge(runs: array of Run, config: ExternalSortConfi
g, numTapes: integer)
    // Розподіляємо runs за стрічками використовуючи числа Фібоначчі
    tapes = array[numTapes] of array of Run
    fibonacci = computeFibonacciDistribution(runs.size(), numTapes)

    // Розподіл початкових runs
    distributeRuns(runs, tapes, fibonacci)

    // Полі-фазне злиття
    while not isComplete(tapes) do
        // Знаходимо стрічки для злиття
        sourceTapes = selectSourceTapes(tapes)
        destinationTape = selectDestinationTape(tapes)

        // Виконуємо злиття
        mergedRun = mergeFromTapes(sourceTapes, config)
        destinationTape.append(mergedRun)

        // Оновлюємо стан стрічок
        updateTapeStates(tapes)
```

Аналіз складності

Для зовнішнього сортування злиттям маємо наступні характеристики:

- I/O складність: $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ операцій введення-виведення

- Кількість проходів: $\lceil\log_k\lceil N/M\rceil\rceil$, де $k = M/B - 1$

- Використання пам’яті: $O(M)$ елементів

Область застосування

Зовнішнє сортування злиттям широко використовується в багатьох областях:

- Системи управління базами даних: Сортування великих таблиць, створення індексів, виконання операцій JOIN та GROUP BY.

- Обробка великих даних: Попередня обробка даних для MapReduce завдань, підготовка даних для аналітики.

- Біоінформатика: Сортування геномних послідовностей, обробка результатів секвенування.

- Фінансові системи: Обробка транзакцій, підготовка звітів, аналіз історичних даних.

- Пошукові системи: Побудова інвертованих індексів, сортування результатів пошуку.

- Наукові обчислення: Обробка результатів моделювання, підготовка даних для візуалізації.

Приклади застосування

Приклад 1: Базове зовнішнє сортування

Розглянемо сортування файлу з 25 числами при обмеженій пам’яті M=5 елементів:

```
Вхідні дані: [64, 34, 25, 12, 22, 11, 90, 88, 53, 42,
              17, 29, 84, 67, 45, 38, 73, 21, 61, 92,
              14, 55, 78, 31, 49]

Параметри:
- M = 5 (розмір пам'яті)
- B = 2 (розмір блоку)

Фаза 1: Створення початкових runs

Run 0: [64, 34, 25, 12, 22] → сортуємо → [12, 22, 25, 34, 64]
Run 1: [11, 90, 88, 53, 42] → сортуємо → [11, 42, 53, 88, 90]
Run 2: [17, 29, 84, 67, 45] → сортуємо → [17, 29, 45, 67, 84]
Run 3: [38, 73, 21, 61, 92] → сортуємо → [21, 38, 61, 73, 92]
Run 4: [14, 55, 78, 31, 49] → сортуємо → [14, 31, 49, 55, 78]

Фаза 2: Злиття

Прохід 1: k = 2 (можемо зливати по 2 runs одночасно)
Merge(Run0, Run1): [11, 12, 22, 25, 34, 42, 53, 64, 88, 90]
Merge(Run2, Run3): [17, 21, 29, 38, 45, 61, 67, 73, 84, 92]
Run4 залишається: [14, 31, 49, 55, 78]
```

```
Прохід 2:
Merge(Merged1, Merged2):
[11, 12, 17, 21, 22, 25, 29, 34, 38, 42, 45, 53, 61, 64, 67, 73, 84,
88, 90, 92]

Прохід 3:
Merge(Merged3, Run4):
[11, 12, 14, 17, 21, 22, 25, 29, 31, 34, 38, 42, 45, 49, 53, 55, 61,
64, 67, 73, 78, 84, 88, 90, 92]

Результат: повністю відсортований файл
```

Приклад 2: Сортування логів веб-сервера

Система для сортування логів веб-сервера за часовою міткою:

```
struct LogEntry {
    timestamp: integer
    ipAddress: string
    method: string
    url: string
    statusCode: integer
    responseSize: integer
}

function sortWebLogs(inputLogFile: string, outputLogFile: string, ava
ilableMemoryMB: integer)
    config = ExternalSortConfig(
        memorySize = availableMemoryMB * 1024 * 1024 / sizeof(LogEntr
y),
        blockSize = 4096 / sizeof(LogEntry),
        inputFile = inputLogFile,
        outputFile = outputLogFile,
        tempDirectory = "/tmp/logsort",
        comparator = compareByTimestamp
    )

    // Парсимо та створюємо початкові runs
    runs = createLogRuns(config)

    // Виконуємо злиття
    multiWayMerge(runs, config)

    // Створюємо індекси для швидкого пошуку
    createTimeIndex(outputLogFile)

function createLogRuns(config: ExternalSortConfig): array of Run
```

```
    runs = []
    runNumber = 0
    inputStream = openTextInputStream(config.inputFile)

    while not inputStream.isEOF() do
        buffer = []

        // Читаємо та парсимо логи
        while buffer.size() < config.memorySize and not inputStream.i
sEOF() do
            line = inputStream.readLine()
            if line != null then
                entry = parseLogEntry(line)
                buffer.append(entry)

        // Сортуємо за timestamp
        buffer.sort(config.comparator)

        // Записуємо run
        runFileName = config.tempDirectory + "/logrun_" + runNumber
        writeBinaryRun(runFileName, buffer)

        runs.append(Run(runFileName, buffer.size(), 0))
        runNumber += 1

    return runs

function compareByTimestamp(entry1: LogEntry, entry2: LogEntry): inte
ger
    return entry1.timestamp - entry2.timestamp
```

Приклад 3: Сортування геномних даних

Обробка великих файлів з геномними послідовностями:

```
struct GenomicRead {
    sequenceId: string
    sequence: string
    quality: string
    position: integer
}

function sortGenomicData(fastqFile: string, outputFile: string, sortB
y: string)
    // Визначаємо функцію порівняння
    comparator = null
    if sortBy == "position" then
```

```
        comparator = compareByPosition
    else if sortBy == "sequence" then
        comparator = compareBySequence
    else if sortBy == "quality" then
        comparator = compareByAverageQuality

    config = ExternalSortConfig(
        memorySize = 4 * 1024 * 1024 * 1024 / estimateReadSize(), //
4GB
        blockSize = 64 * 1024 / estimateReadSize(),
        inputFile = fastqFile,
        outputFile = outputFile,
        tempDirectory = "/scratch/genomicsort",
        comparator = comparator
    )

    // Використовуємо replacement selection для довших runs
    runs = replacementSelectionForGenomics(config)

    // Паралельне злиття для прискорення
    parallelMultiWayMerge(runs, config, numCores = 8)

function replacementSelectionForGenomics(config: ExternalSortConfig):
array of Run
    runs = []
    heap = MinHeap(config.comparator)
    reader = FastqReader(config.inputFile)

    // Оптимізація: використовуємо bloom filter для дедуплікації
    bloomFilter = BloomFilter(expectedElements = 100000000, falsePosi
tiveRate = 0.01)

    currentRun = null
    runNumber = 0
    lastOutputRead = null

    // Заповнюємо heap
    while heap.size() < config.memorySize and reader.hasNext() do
        read = reader.nextRead()
        if not bloomFilter.contains(read.sequenceId) then
            bloomFilter.add(read.sequenceId)
            heap.insert(read)

    while not heap.isEmpty() do
        minRead = heap.extractMin()
```

```
        if currentRun == null then
            currentRun = createGenomicRun(config, runNumber++)

        writeGenomicRead(currentRun, minRead)
        lastOutputRead = minRead

        if reader.hasNext() then
            newRead = reader.nextRead()

            // Перевіряємо дублікати
            if not bloomFilter.contains(newRead.sequenceId) then
                bloomFilter.add(newRead.sequenceId)

                if config.comparator(newRead, lastOutputRead) >= 0 th
en
                    heap.insert(newRead)
                else
                    // Завершуємо поточний run
                    finishRun(currentRun)
                    runs.append(currentRun)
                    currentRun = null
                    heap.insert(newRead)

    return runs
```

Приклад 4: Розподілене зовнішнє сортування

Сортування даних розподілених між кількома вузлами:

```
struct DistributedExternalSort {
    nodes: array of ComputeNode
    coordinator: CoordinatorNode
    partitioner: Partitioner
}

function distributedSort(inputFiles: array of string, outputDirectory
: string)
    // Фаза 1: Локальне сортування на кожному вузлі
    localSortTasks = []

    for i = 0 to inputFiles.size() - 1 do
        nodeIndex = i mod nodes.size()
        task = LocalSortTask(
            inputFile = inputFiles[i],
            node = nodes[nodeIndex],
            outputPrefix = "local_sorted_" + i
```

```
        )
        localSortTasks.append(task)

    // Виконуємо локальне сортування паралельно
    executeInParallel(localSortTasks)

    // Фаза 2: Sampling для визначення розподілу
    samples = collectSamples(localSortTasks, samplingRate = 0.01)
    pivots = determinePivots(samples, nodes.size() - 1)

    // Фаза 3: Партиціонування
    partitionTasks = []
    for task in localSortTasks do
        for pivotIndex = 0 to pivots.size() do
            partitionTask = PartitionTask(
                sourceFile = task.outputFile,
                destinationNode = nodes[pivotIndex],
                minKey = pivots[pivotIndex - 1] if pivotIndex > 0 els
e -∞,
                maxKey = pivots[pivotIndex] if pivotIndex < pivots.si
ze() else +∞
            )
            partitionTasks.append(partitionTask)

    executePartitioning(partitionTasks)

    // Фаза 4: Фінальне злиття на кожному вузлі
    mergeTasks = []
    for node in nodes do
        task = FinalMergeTask(
            node = node,
            inputPartitions = getPartitionsForNode(node),
            outputFile = outputDirectory + "/part_" + node.id
        )
        mergeTasks.append(task)

    executeInParallel(mergeTasks)

function executeLocalSort(task: LocalSortTask)
    config = ExternalSortConfig(
        memorySize = task.node.availableMemory / sizeof(Element),
        blockSize = task.node.optimalIOSize / sizeof(Element),
        inputFile = task.inputFile,
        outputFile = task.outputPrefix + "_sorted",
        tempDirectory = task.node.tempDirectory,
```

```
        comparator = globalComparator
    )

    // Використовуємо всі ядра вузла
    runs = parallelCreateRuns(config, task.node.numCores)

    // Оптимізоване злиття з використанням RAID-0
    optimizedMerge(runs, config, task.node.diskArray)
```

Приклад 5: Потокове зовнішнє сортування

Сортування даних що постійно надходять:

```
struct StreamingExternalSort {
    currentBatch: array of elements
    batchSize: integer
    sortedBatches: PriorityQueue<SortedBatch>
    outputStream: OutputStream
    lastOutputValue: element
}

function initStreamingSort(batchSizeMB: integer, outputFile: string):
StreamingExternalSort
    return StreamingExternalSort(
        currentBatch = [],
        batchSize = batchSizeMB * 1024 * 1024 / sizeof(element),
        sortedBatches = PriorityQueue(compareBatchesByMinElement),
        outputStream = openOutputStream(outputFile),
        lastOutputValue = null
    )

function processStreamElement(sort: StreamingExternalSort, element: e
lement)
    sort.currentBatch.append(element)

    if sort.currentBatch.size() >= sort.batchSize then
        processBatch(sort)

function processBatch(sort: StreamingExternalSort)
    // Сортуємо поточний batch
    inMemorySort(sort.currentBatch, defaultComparator)

    // Створюємо sorted batch об'єкт
    batch = SortedBatch(
        elements = sort.currentBatch,
        currentIndex = 0,
        minElement = sort.currentBatch[0],
```

```
        maxElement = sort.currentBatch[sort.currentBatch.size() - 1]
    )

    // Додаємо до черги
    sort.sortedBatches.insert(batch)

    // Очищаємо поточний batch
    sort.currentBatch = []

    // Виводимо елементи що точно менші за всі майбутні
    outputReadyElements(sort)

function outputReadyElements(sort: StreamingExternalSort)
    while not sort.sortedBatches.isEmpty() do
        minBatch = sort.sortedBatches.peek()

        // Перевіряємо чи можемо виводити елементи
        canOutput = true
        for batch in sort.sortedBatches do
            if batch != minBatch and batch.minElement < getCurrentEle
ment(minBatch) then
                canOutput = false
                break

        if not canOutput then
            break

        // Виводимо елемент
        element = getCurrentElement(minBatch)
        writeElement(sort.outputStream, element)
        sort.lastOutputValue = element

        // Переміщуємо індекс
        minBatch.currentIndex += 1

        // Якщо batch вичерпаний, видаляємо
        if minBatch.currentIndex >= minBatch.elements.size() then
            sort.sortedBatches.remove(minBatch)
        else
            // Оновлюємо позицію в черзі
            sort.sortedBatches.updateKey(minBatch)

function finalizeStreamingSort(sort: StreamingExternalSort)
    // Обробляємо останній неповний batch
    if not sort.currentBatch.isEmpty() then
```

```
        processBatch(sort)

    // Виводимо всі залишкові елементи
    while not sort.sortedBatches.isEmpty() do
        batch = sort.sortedBatches.extractMin()

        while batch.currentIndex < batch.elements.size() do
            element = batch.elements[batch.currentIndex]
            writeElement(sort.outputStream, element)
            batch.currentIndex += 1

    closeOutputStream(sort.outputStream)
```

Переваги та недоліки

Переваги

- Масштабованість: Може обробляти файли будь-якого розміру, обмежені лише доступним дисковим простором.

- Ефективність I/O: Використовує послідовний доступ до диску, що значно швидше за випадковий.

- Передбачуваність: Має гарантовану верхню межу складності O(N log N).

- Стабільність: Зберігає відносний порядок рівних елементів.

- Простота реалізації: Базовий алгоритм відносно простий для розуміння та реалізації.

- Паралелізація: Легко розпаралелюється на етапах створення runs та злиття.

Недоліки

- Використання додаткового простору: Потребує додатковий дисковий простір для тимчасових файлів (до 2N).

- Множинні проходи: Дані читаються та записуються кілька разів.

- Залежність від I/O: Продуктивність сильно залежить від швидкості дискової підсистеми.

- Неефективність для малих даних: Має значні накладні витрати для даних, що поміщаються в пам’ять.

- Складність оптимізації: Оптимальне налаштування параметрів залежить від конкретної системи.

- Фрагментація: Може призводити до фрагментації файлової системи при інтенсивному використанні.

Контрольні запитання

1. Чому традиційні алгоритми сортування неефективні для даних що не поміщаються в пам’ять?

2. Які основні фази алгоритму зовнішнього сортування злиттям?

3. Як обчислюється кількість початкових runs при заданому розмірі пам’яті M та розмірі даних N?

4. Що таке k-шляхове злиття та як вибирається оптимальне значення k?

5. Яка роль купи (heap) в алгоритмі k-шляхового злиття?

6. Як працює заміщувальне сортування (replacement selection)?

7. Чому важливий розмір блоку B при аналізі I/O складності?

8. Які переваги полі-фазного сортування над простим багатошляховим злиттям?

9. Як можна адаптувати алгоритм для роботи з даними змінної довжини?

10. Які стратегії використовуються для мінімізації кількості I/O операцій?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму зовнішнього сортування злиттям

2. Мета

Вивчити принципи роботи з зовнішньою пам’яттю, реалізувати ефективний алгоритм зовнішнього сортування, дослідити вплив параметрів на продуктивність, порівняти різні стратегії оптимізації.

3. Завдання

Реалізувати базовий алгоритм зовнішнього сортування злиттям, додати оптимізації (replacement selection, буферизація), провести експерименти з різними розмірами даних та параметрами, візуалізувати процес роботи алгоритму.

4. Опис

Лабораторна робота включає поетапну реалізацію компонентів зовнішнього сортування: створення початкових runs, k-шляхове злиття, управління тимчасовими файлами, оптимізації для покращення продуктивності.

5. Порядок виконання

- Реалізувати функцію створення початкових відсортованих runs

- Створити ефективну реалізацію k-шляхового злиття з використанням купи

- Додати буферизацію для оптимізації I/O операцій

- Реалізувати replacement selection для створення довших runs

- Створити систему управління тимчасовими файлами

- Додати підтримку різних типів даних через шаблони/generics

- Реалізувати візуалізацію процесу сортування

- Провести benchmarks для різних конфігурацій

- Порівняти з внутрішнім сортуванням для різних розмірів даних

- Дослідити вплив розміру блоку та буфера на продуктивність

6. Результат

Працююча система зовнішнього сортування з графічним інтерфейсом для візуалізації, детальна статистика I/O операцій, графіки залежності продуктивності від параметрів, рекомендації щодо оптимального налаштування для різних сценаріїв використання.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Модуль створення початкових runs з різними стратегіями

- Реалізація k-шляхового злиття з мін-купою

- Система буферизації I/O операцій

- Модуль управління тимчасовими файлами

- Інтерфейс для різних типів даних

2. Продемонструвати роботу додатку:

- Сортування файлів різного розміру (більших за доступну пам’ять)

- Візуалізація етапів створення runs та злиття

- Статистика I/O операцій та використання пам’яті

- Порівняння різних стратегій та параметрів

- Обробка помилок (недостатньо місця на диску, збої)

3. Бути готовим відповісти на питання:

- Як вибрати оптимальний розмір буфера для конкретної системи?

- Коли replacement selection дає найбільший виграш?

- Як мінімізувати кількість проходів злиття?

- Які компроміси між використанням пам’яті та кількістю I/O?

Тести

1. Скільки початкових runs створюється для файлу розміром 1GB при доступній пам’яті 100MB?

- 1) 10

- 2) 100

- 3) 1000

- 4) 1

2. Що є основною метрикою ефективності для алгоритмів зовнішньої пам’яті?

- 1) Час виконання

- 2) Використання CPU

- 3) Кількість I/O операцій

- 4) Використання пам’яті

3. Яка структура даних використовується для ефективного k-шляхового злиття?

- 1) Стек

- 2) Черга

- 3) Мін-купа

- 4) Хеш-таблиця

4. Що таке replacement selection?

- 1) Заміна елементів в масиві

- 2) Метод створення довших початкових runs

- 3) Алгоритм вибору pivots

- 4) Стратегія кешування

5. Яка I/O складність зовнішнього сортування злиттям?

- 1) O(N)

- 2) O(N log N)

- 3) O(N/B log(N/B) / log(M/B))

- 4) $O(N^2)$

Відповіді:

1-1,

2-3,

3-3,

4-2,

5-3

Тести підвищеної складності

1. Чому послідовний доступ ефективніший за випадковий на жорстких дисках?

- 1) Менше використання CPU

- 2) Відсутність механічного переміщення головки

- 3) Краще стиснення даних

- 4) Швидший кеш

2. Як replacement selection може створити run довжиною 2M з пам’яттю розміром M?

- 1) Використовує стиснення

- 2) Зберігає в купі елементи ⩾ останнього виведеного

- 3) Подвоює розмір пам’яті

- 4) Читає дані двічі

3. Що визначає оптимальне значення k для k-шляхового злиття?

- 1) Розмір файлу

- 2) Швидкість CPU

- 3) Доступна пам’ять та розмір блоку

- 4) Кількість ядер процесора

4. Яка перевага полі-фазного сортування?

- 1) Простіша реалізація

- 2) Менша кількість проходів злиття

- 3) Не потребує додаткового місця

- 4) Швидше створення runs

5. Коли зовнішнє сортування стає ефективнішим за внутрішнє?

- 1) Завжди
- 2) Коли дані не поміщаються в RAM
- 3) Для малих файлів
- 4) При використанні SSD

6. Як мінімізувати write amplification при зовнішньому сортуванні?

- 1) Використовувати більші блоки
- 2) Зменшити кількість проходів злиття
- 3) Збільшити розмір пам’яті
- 4) Всі відповіді правильні

7. Що таке “snowplow effect” в контексті replacement selection?

- 1) Накопичення малих елементів в кінці
- 2) Збільшення швидкості з часом
- 3) Зменшення розміру runs
- 4) Блокування I/O операцій

8. Як впливає розмір блоку B на продуктивність?

- 1) Не впливає
- 2) Більший B - менше I/O операцій але більше пам’яті
- 3) Менший B завжди кращий
- 4) Залежить від CPU

9. Чому важливо видаляти тимчасові файли після кожного проходу?

- 1) Безпека даних
- 2) Економія дискового простору
- 3) Швидкість доступу
- 4) Запобігання помилкам

10. Як адаптувати алгоритм для SSD накопичувачів?

- 1) Збільшити паралелізм I/O
- 2) Зменшити розмір блоку
- 3) Використовувати випадковий доступ

- 4) Відключити буферизацію

Відповіді:

1. 2) Механічні диски мають затримку на переміщення головки
2. 2) Купа дозволяє вивести більше елементів ніж початковий розмір
3. 3) k = (M/B) - 1 для оптимального використання пам’яті
4. 2) Використовує числа Фібоначчі для оптимального розподілу
5. 2) Overhead зовнішнього сортування виправданий лише для великих даних
6. 4) Всі методи допомагають зменшити кількість перезаписів
7. 1) Малі елементи “виштовхуються” в наступний run
8. 2) Баланс між кількістю I/O та використанням пам’яті
9. 2) Потрібно до 2× простору від розміру даних
10. 1) SSD підтримують паралельні операції без втрати швидкості

Завдання для самостійної роботи

1. Реалізуйте адаптивний алгоритм що автоматично вибирає між внутрішнім та зовнішнім сортуванням.
2. Створіть версію з підтримкою стиснення даних для економії I/O.
3. Розробіть паралельну версію що використовує кілька дисків одночасно.
4. Реалізуйте cascade merge sort та порівняйте з базовим алгоритмом.
5. Створіть версію для сортування даних з змінною довжиною записів.
6. Розробіть систему checkpointing для відновлення після збоїв.
7. Реалізуйте гібридний алгоритм що комбінує quicksort та merge sort.
8. Створіть розподілену версію для кластера комп’ютерів.
9. Розробіть спеціалізовану версію для часових рядів.
10. Дослідіть використання GPU для прискорення фази створення runs.

Теми лабораторних робіт

1. Базова реалізація зовнішнього сортування з візуалізацією.
2. Порівняльний аналіз стратегій створення початкових runs.
3. Оптимізація I/O операцій через буферизацію та передбачення.
4. Реалізація полі-фазного сортування.

5. Розробка системи для сортування баз даних.

6. Створення утиліти сортування з підтримкою різних форматів.

7. Паралельне зовнішнє сортування на багатоядерних системах.

8. Адаптація алгоритму для різних типів накопичувачів.

9. Інтеграція зовнішнього сортування в СУБД.

10. Дослідження енергоефективності різних стратегій сортування.

Cache-oblivious алгоритми

Теоретичні відомості

Cache-oblivious алгоритми представляють революційний підхід до проектування алгоритмів, який дозволяє досягати оптимальної продуктивності на різних рівнях ієрархії пам’яті без явного знання параметрів кешу. Ця парадигма була введена Гаральдом Прокопом у його магістерській дисертації в MIT у 1999 році під керівництвом Чарльза Лейзерсона.

Фундаментальні концепції

Сучасні комп’ютерні системи мають складну ієрархію пам’яті: регістри процесора, кілька рівнів кешу (L1, L2, L3), оперативна пам’ять, SSD, жорсткі диски. Кожен наступний рівень має більшу ємність, але й більшу затримку доступу. Наприклад, доступ до L1 кешу займає 1-2 такти процесора, до L2 - 10-20 тактів, до L3 - 30-70 тактів, а до оперативної пам’яті - сотні тактів.

Традиційні cache-aware алгоритми оптимізуються під конкретні параметри кешу: розмір блоку B та загальний розмір кешу M. Проблема такого підходу полягає в тому, що ці параметри різняться між системами, і алгоритм, оптимізований для однієї конфігурації, може погано працювати на іншій.

Cache-oblivious алгоритми вирішують цю проблему, використовуючи рекурсивні структури та алгоритми, які автоматично адаптуються до будь-якої ієрархії пам’яті. Ключова ідея полягає в тому, що правильно спроектований рекурсивний алгоритм буде ефективно використовувати кеш на всіх рівнях ієрархії одночасно.

Модель обчислень

Cache-oblivious модель базується на наступних припущеннях:

- **Ідеальна модель кешу**: Система має два рівні пам’яті - швидкий кеш розміром M та повільну основну пам’ять необмеженого розміру

- **Автоматичне управління**: Система автоматично переміщує блоки розміром B між кешем та основною пам’яттю

- **Оптимальна заміна**: Використовується оптимальна офлайн стратегія заміни блоків (алгоритм Беледі)

- **Tall cache assumption**: $M = \Omega(B^2)$, тобто кеш може вмістити принаймні B блоків

Важливо, що алгоритм не знає значення M та B, але повинен бути ефективним для всіх можливих значень цих параметрів.

Cache-oblivious матричне транспонування

Розглянемо один з найпростіших cache-oblivious алгоритмів - транспонування матриці. Наївний алгоритм має поганий шаблон доступу до пам’яті при читанні по стовпцях.

```
struct Matrix {
    data: array[n][n] of element
    rowStart: integer
    colStart: integer
    rows: integer
    cols: integer
}

function naiveTranspose(A: Matrix, B: Matrix)
    // Наївний алгоритм - погана локальність для великих матриць
    for i = 0 to A.rows - 1 do
        for j = 0 to A.cols - 1 do
            B.data[j][i] = A.data[i][j]

function cacheObliviousTranspose(A: Matrix, B: Matrix)
    // База рекурсії - маленькі блоки транспонуємо напряму
    if A.rows <= CUTOFF and A.cols <= CUTOFF then
        for i = A.rowStart to A.rowStart + A.rows - 1 do
            for j = A.colStart to A.colStart + A.cols - 1 do
                B.data[j - A.colStart + B.rowStart][i - A.rowStart +
B.colStart] =
                    A.data[i][j]
        return

    // Рекурсивний поділ по довшому виміру
    if A.rows >= A.cols then
```

```
    // Ділимо по рядках
    mid = A.rows / 2

    A1 = Matrix(
        data = A.data,
        rowStart = A.rowStart,
        colStart = A.colStart,
        rows = mid,
        cols = A.cols
    )

    A2 = Matrix(
        data = A.data,
        rowStart = A.rowStart + mid,
        colStart = A.colStart,
        rows = A.rows - mid,
        cols = A.cols
    )

    B1 = Matrix(
        data = B.data,
        rowStart = B.rowStart,
        colStart = B.colStart,
        rows = B.cols,
        cols = mid
    )

    B2 = Matrix(
        data = B.data,
        rowStart = B.rowStart,
        colStart = B.colStart + mid,
        rows = B.cols,
        cols = A.rows - mid
    )

    // Рекурсивно транспонуємо підматриці
    cacheObliviousTranspose(A1, B1)
    cacheObliviousTranspose(A2, B2)
else
    // Ділимо по стовпцях
    mid = A.cols / 2

    A1 = Matrix(
        data = A.data,
        rowStart = A.rowStart,
```

```
            colStart = A.colStart,
            rows = A.rows,
            cols = mid
        )

        A2 = Matrix(
            data = A.data,
            rowStart = A.rowStart,
            colStart = A.colStart + mid,
            rows = A.rows,
            cols = A.cols - mid
        )

        B1 = Matrix(
            data = B.data,
            rowStart = B.rowStart,
            colStart = B.colStart,
            rows = mid,
            cols = B.rows
        )

        B2 = Matrix(
            data = B.data,
            rowStart = B.rowStart + mid,
            colStart = B.colStart,
            rows = A.cols - mid,
            cols = B.rows
        )

        cacheObliviousTranspose(A1, B1)
        cacheObliviousTranspose(A2, B2)
```

Cache-oblivious матричне множення

Матричне множення є класичним прикладом алгоритму, де cache-oblivious підхід показує значні переваги. Розглянемо алгоритм множення матриць $C = A \times B$.

```
function cacheObliviousMatMul(A: Matrix, B: Matrix, C: Matrix)
    // База рекурсії
    if A.rows <= CUTOFF and A.cols <= CUTOFF and B.cols <= CUTOFF the
n
        // Звичайне множення для малих матриць
        for i = 0 to A.rows - 1 do
            for j = 0 to B.cols - 1 do
                sum = 0
```

```
                for k = 0 to A.cols - 1 do
                    sum += A.data[A.rowStart + i][A.colStart + k] *
                           B.data[B.rowStart + k][B.colStart + j]
                C.data[C.rowStart + i][C.colStart + j] += sum
        return

    // Вибираємо найбільший вимір для поділу
    if A.rows >= max(A.cols, B.cols) then
        // Ділимо A по рядках
        mid = A.rows / 2

        A1 = submatrix(A, 0, 0, mid, A.cols)
        A2 = submatrix(A, mid, 0, A.rows - mid, A.cols)
        C1 = submatrix(C, 0, 0, mid, C.cols)
        C2 = submatrix(C, mid, 0, C.rows - mid, C.cols)

        cacheObliviousMatMul(A1, B, C1)
        cacheObliviousMatMul(A2, B, C2)

    else if B.cols >= max(A.rows, A.cols) then
        // Ділимо B по стовпцях
        mid = B.cols / 2

        B1 = submatrix(B, 0, 0, B.rows, mid)
        B2 = submatrix(B, 0, mid, B.rows, B.cols - mid)
        C1 = submatrix(C, 0, 0, C.rows, mid)
        C2 = submatrix(C, 0, mid, C.rows, C.cols - mid)

        cacheObliviousMatMul(A, B1, C1)
        cacheObliviousMatMul(A, B2, C2)

    else
        // Ділимо по спільному виміру
        mid = A.cols / 2

        A1 = submatrix(A, 0, 0, A.rows, mid)
        A2 = submatrix(A, 0, mid, A.rows, A.cols - mid)
        B1 = submatrix(B, 0, 0, mid, B.cols)
        B2 = submatrix(B, mid, 0, B.rows - mid, B.cols)

        cacheObliviousMatMul(A1, B1, C)
        cacheObliviousMatMul(A2, B2, C)

function submatrix(M: Matrix, rowOffset: integer, colOffset: integer,
                   rows: integer, cols: integer): Matrix
```

```
    return Matrix(
        data = M.data,
        rowStart = M.rowStart + rowOffset,
        colStart = M.colStart + colOffset,
        rows = rows,
        cols = cols
    )
```

Cache-oblivious сортування: Funnel Sort

Funnel Sort є cache-oblivious алгоритмом сортування, який досягає оптимальної складності $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ cache misses. Алгоритм використовує спеціальну структуру даних - k-funnel.

```
struct Funnel {
    k: integer                      // кількість вхідних потоків
    buffers: array[k] of Buffer     // вхідні буфери
    merger: Merger                  // структура для злиття
    output: Buffer                  // вихідний буфер
}

struct Buffer {
    capacity: integer
    size: integer
    data: array of element
    front: integer
    rear: integer
}

function funnelSort(A: array of element, n: integer): array of elemen
t
    if n <= SMALL_THRESHOLD then
        // Використовуємо звичайне сортування для малих масивів
        return insertionSort(A, n)

    // Обчислюємо оптимальне k
    k = computeOptimalK(n)

    // Рекурсивно сортуємо підмасиви
    subarrays = []
    subsize = n / k

    for i = 0 to k - 1 do
        start = i * subsize
        end = min((i + 1) * subsize, n)
        subarray = funnelSort(A[start:end], end - start)
```

```
        subarrays.append(subarray)

    // Створюємо k-funnel для злиття
    funnel = createKFunnel(k, n)

    // Завантажуємо відсортовані підмасиви у funnel
    for i = 0 to k - 1 do
        loadIntoFunnel(funnel, i, subarrays[i])

    // Виконуємо злиття
    result = []
    while funnel.hasMore() do
        element = funnel.extract()
        result.append(element)

    return result

function createKFunnel(k: integer, totalElements: integer): Funnel
    // Створюємо рекурсивну структуру funnel
    if k <= FUNNEL_BASE_SIZE then
        // Базовий випадок - простий merger
        return createBaseFunnel(k)

    // Рекурсивна побудова
    sqrtK = ceil(sqrt(k))

    // Створюємо буфери
    buffers = []
    bufferSize = computeBufferSize(k, totalElements)

    for i = 0 to k - 1 do
        buffer = Buffer(
            capacity = bufferSize,
            size = 0,
            data = array[bufferSize],
            front = 0,
            rear = 0
        )
        buffers.append(buffer)

    // Створюємо рекурсивну структуру merger
    topFunnel = createKFunnel(sqrtK, totalElements / sqrtK)
    bottomFunnels = []

    for i = 0 to sqrtK - 1 do
```

```
        bottomFunnel = createKFunnel(sqrtK, totalElements / k)
        bottomFunnels.append(bottomFunnel)

    merger = Merger(
        topFunnel = topFunnel,
        bottomFunnels = bottomFunnels,
        connections = setupConnections(sqrtK)
    )

    outputBuffer = Buffer(
        capacity = bufferSize * sqrtK,
        size = 0,
        data = array[bufferSize * sqrtK],
        front = 0,
        rear = 0
    )

    return Funnel(k, buffers, merger, outputBuffer)

function extractFromFunnel(funnel: Funnel): element
    // Перевіряємо чи є елементи в output buffer
    if funnel.output.size == 0 then
        fillOutputBuffer(funnel)

    // Витягуємо елемент з output buffer
    element = funnel.output.data[funnel.output.front]
    funnel.output.front = (funnel.output.front + 1) % funnel.output.c
apacity
    funnel.output.size -= 1

    return element

function fillOutputBuffer(funnel: Funnel)
    // Рекурсивно заповнюємо output buffer
    while funnel.output.size < funnel.output.capacity / 2 do
        // Отримуємо наступний елемент від merger
        element = extractFromMerger(funnel.merger)

        if element == null then
            break  // Більше немає елементів

        // Додаємо до output buffer
        funnel.output.data[funnel.output.rear] = element
        funnel.output.rear = (funnel.output.rear + 1) % funnel.output
```

```
.capacity
        funnel.output.size += 1
```

Компоновка Ван Емде Боаса

Компоновка Ван Емде Боаса є cache-oblivious способом розміщення дерева в пам'яті, який мінімізує промахи в кеші при обході дерева.

```
function buildVEBLayout(tree: BinaryTree, n: integer): array of Node
    // Алокуємо масив для всіх вузлів
    layout = array[n] of Node
    position = 0

    // Рекурсивно розміщуємо вузли
    vebLayoutRecursive(tree.root, layout, position, n)

    return layout

function vebLayoutRecursive(node: TreeNode, layout: array,
                            position: ref integer, size: integer)
    if node == null or size == 0 then
        return

    if size == 1 then
        // Базовий випадок - один вузол
        layout[position] = node
        position += 1
        return

    // Обчислюємо розмір верхньої та нижньої частин
    h = floor(log2(size + 1))  // висота повного дерева
    topHeight = ceil(h / 2)
    bottomHeight = h - topHeight

    topSize = (1 << topHeight) - 1  // 2^topHeight - 1
    bottomSize = size - topSize

    // Розміщуємо верхню частину дерева
    topNodes = extractTopSubtree(node, topHeight)
    vebLayoutRecursive(topNodes.root, layout, position, topSize)

    // Розміщуємо нижні піддерева
    for subtree in topNodes.bottomSubtrees do
        subtreeSize = min((1 << bottomHeight) - 1,
                          remainingNodes(subtree, bottomHeight))
        vebLayoutRecursive(subtree, layout, position, subtreeSize)
```

```
function searchVEBTree(layout: array of Node, key: Key, n: integer):
Value or null
    index = 0
    size = n

    while index < n and size > 0 do
        node = layout[index]

        if node.key == key then
            return node.value

        if size == 1 then
            return null  // не знайдено

        // Обчислюємо позиції дітей у VEB layout
        h = floor(log2(size + 1))
        topHeight = ceil(h / 2)
        topSize = (1 << topHeight) - 1

        if key < node.key then
            // Йдемо в ліве піддерево
            leftSubtreeStart = index + topSize
            leftSubtreeSize = computeLeftSubtreeSize(size, topSize)

            index = leftSubtreeStart
            size = leftSubtreeSize
        else
            // Йдемо в праве піддерево
            leftSubtreeSize = computeLeftSubtreeSize(size, topSize)
            rightSubtreeStart = index + topSize + leftSubtreeSize
            rightSubtreeSize = size - topSize - leftSubtreeSize

            index = rightSubtreeStart
            size = rightSubtreeSize

    return null
```

Cache-oblivious пріоритетна черга

Cache-oblivious пріоритетна черга використовує модифіковану структуру бінарної купи з cache-oblivious layout.

```
struct CacheObliviousPQ {
    heap: array of element
    size: integer
    capacity: integer
}
```

```
function createCacheObliviousPQ(initialCapacity: integer): CacheObliv
iousPQ
    // Створюємо купу з VEB layout
    capacity = nextPowerOf2(initialCapacity)
    heap = array[capacity]

    return CacheObliviousPQ(heap, 0, capacity)

function insert(pq: CacheObliviousPQ, element: element)
    if pq.size == pq.capacity then
        resize(pq)

    // Додаємо елемент в кінець
    pq.heap[pq.size] = element
    pq.size += 1

    // Відновлюємо властивість купи
    bubbleUp(pq, pq.size - 1)

function bubbleUp(pq: CacheObliviousPQ, index: integer)
    while index > 0 do
        parentIndex = getParentIndex(index, pq.size)

        if pq.heap[index] <= pq.heap[parentIndex] then
            break

        swap(pq.heap[index], pq.heap[parentIndex])
        index = parentIndex

function extractMax(pq: CacheObliviousPQ): element
    if pq.size == 0 then
        throw EmptyQueueException()

    max = pq.heap[0]
    pq.size -= 1

    if pq.size > 0 then
        pq.heap[0] = pq.heap[pq.size]
        bubbleDown(pq, 0)

    return max

function bubbleDown(pq: CacheObliviousPQ, index: integer)
    while true do
```

```
        leftChild = getLeftChildIndex(index, pq.size)
        rightChild = getRightChildIndex(index, pq.size)
        largest = index

        if leftChild < pq.size and pq.heap[leftChild] > pq.heap[large
st] then
            largest = leftChild

        if rightChild < pq.size and pq.heap[rightChild] > pq.heap[lar
gest] then
            largest = rightChild

        if largest == index then
            break

        swap(pq.heap[index], pq.heap[largest])
        index = largest

// Функції для навігації по VEB layout
function getParentIndex(index: integer, heapSize: integer): integer
    // Обчислюємо позицію батька у VEB layout
    level = getLevel(index, heapSize)
    positionInLevel = getPositionInLevel(index, level, heapSize)

    if level == 0 then
        return -1  // немає батька

    parentLevel = level - 1
    parentPositionInLevel = positionInLevel / 2

    return getIndexFromPosition(parentLevel, parentPositionInLevel, h
eapSize)

function getLeftChildIndex(index: integer, heapSize: integer): intege
r
    level = getLevel(index, heapSize)
    positionInLevel = getPositionInLevel(index, level, heapSize)

    childLevel = level + 1
    leftChildPosition = positionInLevel * 2

    return getIndexFromPosition(childLevel, leftChildPosition, heapSi
ze)
```

Область застосування

Cache-oblivious алгоритми знаходять широке застосування в різних областях:

- Системи баз даних: B-дерева з cache-oblivious layout для індексів, cache-oblivious алгоритми join та sort.

- Наукові обчислення: Матричні операції, розв'язання систем лінійних рівнянь, FFT.

- Обробка зображень: Фільтрація, трансформації, стиснення з ефективним використанням кешу.

- Геометричні алгоритми: Пошук найближчих сусідів, побудова дерев розбиття простору.

- Компілятори: Оптимізація розміщення коду та даних для кращої локальності.

- Операційні системи: Алгоритми планування та управління пам'яттю.

Приклади застосування

Приклад 1: Cache-oblivious транспонування матриці 4×4

Розглянемо транспонування матриці розміром 4×4:

```
Вхідна матриця A:        Вихідна матриця B:
1  2  3  4               1  5  9  13
5  6  7  8               2  6  10 14
9  10 11 12              3  7  11 15
13 14 15 16              4  8  12 16

CUTOFF = 2

Виклик: cacheObliviousTranspose(A[0:4][0:4], B[0:4][0:4])

Крок 1: rows=4 >= cols=4, ділимо по рядках
- A1 = A[0:2][0:4], B1 = B[0:4][0:2]
- A2 = A[2:4][0:4], B2 = B[0:4][2:4]

Крок 2: Рекурсивний виклик для A1, B1
  rows=2 <= CUTOFF, cols=4 > CUTOFF, ділимо по стовпцях
  - A11 = A[0:2][0:2], B11 = B[0:2][0:2]
  - A12 = A[0:2][2:4], B12 = B[2:4][0:2]

  Крок 2.1: A11→B11 (база рекурсії)
    B[0][0] = A[0][0] = 1
    B[1][0] = A[0][1] = 2
    B[0][1] = A[1][0] = 5
```

```
    B[1][1] = A[1][1] = 6

  Крок 2.2: A12→B12 (база рекурсії)
    B[2][0] = A[0][2] = 3
    B[3][0] = A[0][3] = 4
    B[2][1] = A[1][2] = 7
    B[3][1] = A[1][3] = 8

Крок 3: Рекурсивний виклик для A2, B2
  (аналогічно)

Результат: матриця B містить транспоновану матрицю A
```

Приклад 2: Пошук у VEB дереві

Розглянемо бінарне дерево пошуку з 7 вузлами у VEB layout:

```
Логічне дерево:           VEB Layout в масиві:
      4                   [4, 2, 6, 1, 3, 5, 7]
    /   \
   2     6                Індекси: 0  1  2  3  4  5  6
  / \   / \
 1   3 5   7

Пошук ключа 5:
1. index=0, node=4, size=7
   5 > 4, йдемо праворуч
   h=2, topHeight=1, topSize=1
   rightSubtreeStart = 0 + 1 + 3 = 4
   rightSubtreeSize = 3

2. index=4, node=6, size=3
   5 < 6, йдемо ліворуч
   h=1, topHeight=1, topSize=1
   leftSubtreeStart = 4 + 1 = 5
   leftSubtreeSize = 1

3. index=5, node=5, size=1
   5 == 5, знайдено!
```

Приклад 3: Cache-oblivious матричне множення

Множення матриць 4×4:

```
A =  1 2 3 4     B =  1 0 1 0     C = A×B = ?
     5 6 7 8          0 1 0 1
     9 0 1 2          1 0 1 0
     3 4 5 6          0 1 0 1
```

```
CUTOFF = 2

Виклик: cacheObliviousMatMul(A[0:4][0:4], B[0:4][0:4], C[0:4][0:4])

Крок 1: max(4,4,4)=4, всі виміри рівні, ділимо A по рядках
- A1 = A[0:2][0:4], C1 = C[0:2][0:4]
- A2 = A[2:4][0:4], C2 = C[2:4][0:4]

Крок 2: Рекурсія для (A1, B, C1)
  max(2,4,4)=4, ділимо B по стовпцях
  - B1 = B[0:4][0:2], C11 = C[0:2][0:2]
  - B2 = B[0:4][2:4], C12 = C[0:2][2:4]

  Крок 2.1: (A1, B1, C11) - досягли CUTOFF
    C[0][0] += 1×1 + 2×0 + 3×1 + 4×0 = 4
    C[0][1] += 1×0 + 2×1 + 3×0 + 4×1 = 6
    C[1][0] += 5×1 + 6×0 + 7×1 + 8×0 = 12
    C[1][1] += 5×0 + 6×1 + 7×0 + 8×1 = 14

  Крок 2.2: (A1, B2, C12) - аналогічно

Фінальний результат:
C =  4  6  4  6
    12 14 12 14
     9  2  9  2
     7 10  7 10
```

Приклад 4: Funnel Sort для 8 елементів

Сортування масиву [7, 3, 5, 1, 8, 2, 6, 4]:

```
n = 8, k = 2 (оптимальне для 8 елементів)

Крок 1: Рекурсивне сортування підмасивів
- Підмасив 1: [7, 3, 5, 1] → funnelSort → [1, 3, 5, 7]
- Підмасив 2: [8, 2, 6, 4] → funnelSort → [2, 4, 6, 8]

Крок 2: Створення 2-funnel
Структура:
         Funnel
        /      \
   Buffer1    Buffer2
   [1,3,5,7]  [2,4,6,8]
       \        /
        Merger
          |
     Output Buffer
```

```
Крок 3: Злиття
Ітерація 1: min(1,2) = 1 → output: [1]
Ітерація 2: min(3,2) = 2 → output: [1,2]
Ітерація 3: min(3,4) = 3 → output: [1,2,3]
Ітерація 4: min(5,4) = 4 → output: [1,2,3,4]
Ітерація 5: min(5,6) = 5 → output: [1,2,3,4,5]
Ітерація 6: min(7,6) = 6 → output: [1,2,3,4,5,6]
Ітерація 7: min(7,8) = 7 → output: [1,2,3,4,5,6,7]
Ітерація 8: залишилось 8 → output: [1,2,3,4,5,6,7,8]

Результат: [1,2,3,4,5,6,7,8]
```

Приклад 5: Cache-oblivious B-дерево

Реалізація cache-oblivious B-дерева для ефективного пошуку:

```
struct COBTree {
    nodes: array of Node      // вузли у VEB layout
    nodeCount: integer        // кількість вузлів
    height: integer          // висота дерева
    branchingFactor: integer // ступінь вузлів
}

struct Node {
    keys: array of Key
    values: array of Value
    children: array of integer  // індекси дітей у VEB layout
    keyCount: integer
    isLeaf: boolean
}

function insertIntoCOBTree(tree: COBTree, key: Key, value: Value)
    if tree.nodeCount == 0 then
        // Створюємо корінь
        root = Node(
            keys = [key],
            values = [value],
            children = [],
            keyCount = 1,
            isLeaf = true
        )
        tree.nodes[0] = root
        tree.nodeCount = 1
        tree.height = 1
        return
```

```
    // Знаходимо листок для вставки
    path = findInsertionPath(tree, key)
    leafIndex = path[path.length - 1]
    leaf = tree.nodes[leafIndex]

    // Вставляємо у листок
    insertIntoNode(leaf, key, value)

    // Якщо листок переповнений, розщеплюємо
    if leaf.keyCount > tree.branchingFactor then
        splitNode(tree, leafIndex, path)

function findInsertionPath(tree: COBTree, key: Key): array of integer
    path = []
    currentIndex = 0  // корінь завжди на індексі 0

    while not tree.nodes[currentIndex].isLeaf do
        path.append(currentIndex)
        node = tree.nodes[currentIndex]

        // Знаходимо дитину для спуску
        childPosition = 0
        while childPosition < node.keyCount and key > node.keys[child
Position] do
            childPosition += 1

        // Обчислюємо індекс дитини у VEB layout
        childIndex = getChildIndex(currentIndex, childPosition, tree)
        currentIndex = childIndex

    path.append(currentIndex)
    return path

function getChildIndex(parentIndex: integer, childPosition: integer,
                       tree: COBTree): integer
    // Обчислюємо позицію дитини у VEB layout
    level = getNodeLevel(parentIndex, tree)
    subtreeSize = getSubtreeSize(level + 1, tree.height)

    // Базова позиція для дітей цього вузла
    firstChildIndex = parentIndex + 1

    // Пропускаємо попередні піддерева
    for i = 0 to childPosition - 1 do
        firstChildIndex += subtreeSize
```

```
    return firstChildIndex

function searchCOBTree(tree: COBTree, key: Key): Value or null
    currentIndex = 0

    while currentIndex < tree.nodeCount do
        node = tree.nodes[currentIndex]

        // Шукаємо ключ у вузлі
        position = binarySearch(node.keys, key, node.keyCount)

        if position >= 0 then
            // Знайдено
            return node.values[position]

        if node.isLeaf then
            // Не знайдено
            return null

        // Визначаємо дитину для спуску
        insertPosition = -(position + 1)
        currentIndex = getChildIndex(currentIndex, insertPosition, tr
ee)

    return null
```

Переваги та недоліки

Переваги

- Універсальність: Працюють ефективно на будь-якій ієрархії пам’яті без налаштування.

- Портативність: Один код оптимально працює на різних архітектурах.

- Простота: Не потребують знання параметрів системи під час розробки.

- Асимптотична оптимальність: Досягають теоретично оптимальної кількості cache misses.

- Адаптивність: Автоматично адаптуються до змін в системі (різні рівні кешу).

- Елегантність: Часто мають простішу структуру ніж cache-aware аналоги.

Недоліки

- Константи: Можуть мати гірші константи порівняно з оптимізованими cache-aware алгоритмами.

- Складність аналізу: Доведення оптимальності часто складніше ніж для звичайних алгоритмів.

- Overhead рекурсії: Глибока рекурсія може створювати додаткові витрати.

- Не завжди практичні: Для деяких задач cache-aware рішення простіші та ефективніші.

- Обмеження моделі: Модель ідеального кешу не враховує всі особливості реальних систем.

- Важко профілювати: Складніше виміряти та оптимізувати продуктивність.

Контрольні запитання

1. Що таке cache-oblivious алгоритми та чим вони відрізняються від cache-aware?

2. Які основні припущення моделі ідеального кешу?

3. Поясніть принцип роботи cache-oblivious транспонування матриці.

4. Чому рекурсивний поділ забезпечує хорошу локальність на всіх рівнях кешу?

5. Що таке Van Emde Boas layout та як він покращує локальність?

6. Як працює k-funnel у алгоритмі Funnel Sort?

7. Яка асимптотична складність cache-oblivious сортування?

8. Чому важливе припущення tall cache ($M = \Omega(B^2)$)?

9. Як адаптувати існуючі алгоритми до cache-oblivious парадигми?

10. Які практичні обмеження cache-oblivious алгоритмів?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження cache-oblivious алгоритмів

2. Мета

Вивчити принципи проектування cache-oblivious алгоритмів, реалізувати основні алгоритми, дослідити їх продуктивність на різних архітектурах, порівняти з cache-aware та наївними реалізаціями.

3. Завдання

Реалізувати cache-oblivious алгоритми транспонування та множення матриць, створити VEB layout для бінарного дерева, провести експерименти з різними розмірами даних, виміряти кількість cache misses та час виконання.

4. Опис

Лабораторна робота включає реалізацію базових cache-oblivious алгоритмів, вимірювання їх продуктивності за допомогою hardware performance counters, візуалізацію патернів доступу до пам’яті, порівняння з традиційними алгоритмами.

5. Порядок виконання

- Реалізувати наївне та cache-oblivious транспонування матриці
- Виміряти cache misses для різних розмірів матриць
- Реалізувати cache-oblivious матричне множення
- Створити VEB layout для повного бінарного дерева
- Реалізувати пошук у дереві з VEB layout
- Порівняти з звичайним бінарним деревом
- Візуалізувати патерни доступу до пам’яті
- Провести експерименти на різних системах
- Дослідити вплив розміру CUTOFF на продуктивність
- Проаналізувати результати та зробити висновки

6. Результат

Набір cache-oblivious алгоритмів з детальним аналізом продуктивності, графіки залежності cache misses від розміру даних, візуалізація патернів доступу до пам’яті, рекомендації щодо використання cache-oblivious підходу для різних задач.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Реалізація cache-oblivious транспонування
- Cache-oblivious матричне множення
- VEB layout для бінарного дерева
- Система вимірювання performance counters
- Візуалізація патернів доступу

2. Продемонструвати роботу додатку:
- Порівняння продуктивності різних алгоритмів
- Графіки cache misses для різних розмірів
- Візуалізація layout структур даних
- Аналіз впливу параметрів на продуктивність
- Результати на різних архітектурах
3. Бути готовим відповісти на питання:
- Чому рекурсивний поділ покращує локальність?
- Як вибрати оптимальне значення CUTOFF?
- Коли cache-oblivious підхід виграє у cache-aware?
- Як адаптувати інші алгоритми до cache-oblivious моделі?

Тести

1. Що НЕ знає cache-oblivious алгоритм?
- 1) Розмір вхідних даних
- 2) Алгоритм заміни сторінок
- 3) Розмір блоку B та кешу M
- 4) Тип процесора
2. Яка основна ідея cache-oblivious алгоритмів?
- 1) Використання спеціальних інструкцій процесора
- 2) Рекурсивний поділ для адаптації до будь-якого рівня кешу
- 3) Повне ігнорування кешу
- 4) Використання тільки регістрів
3. Що таке VEB layout?
- 1) Спосіб стиснення даних
- 2) Рекурсивне розміщення дерева в пам’яті
- 3) Алгоритм сортування
- 4) Тип кешу процесора
4. Яка складність cache misses для cache-oblivious матричного множення?
- 1) $O(n^3)$
- 2) $O(n^2)$

- 3) $O(n^3/B\sqrt{M})$

- 4) $O(n \log n)$

5. Коли припиняється рекурсія в cache-oblivious алгоритмах?

- 1) Ніколи

- 2) При досягненні базового розміру CUTOFF

- 3) Після 10 рівнів

- 4) Коли закінчується пам’ять

Відповіді:

1-3,

2-2,

3-2,

4-3,

5-2

Тести підвищеної складності

1. Чому tall cache assumption ($M = \Omega(B^2)$) важливе?

- 1) Для ефективності hardware

- 2) Щоб рекурсія могла ефективно використовувати кеш

- 3) Для зменшення споживання енергії

- 4) Це не важливо

2. Як cache-oblivious алгоритми досягають оптимальності на всіх рівнях?

- 1) Використовують prefetching

- 2) Рекурсивний поділ створює задачі всіх розмірів

- 3) Оптимізують під конкретний процесор

- 4) Використовують паралелізм

3. Яка перевага VEB layout над звичайним обходом в глибину?

- 1) Менше пам’яті

- 2) Кращий баланс між вертикальним та горизонтальним доступом

- 3) Простіша реалізація

- 4) Швидша вставка

4. Чому Funnel Sort оптимальний?

- 1) Використовує quicksort
- 2) Мінімізує I/O через оптимальну структуру k-funnel
- 3) Працює in-place
- 4) Паралельний

5. Що відбувається при CUTOFF = 1?
- 1) Максимальна ефективність
- 2) Занадто глибока рекурсія, overhead
- 3) Алгоритм не працює
- 4) Оптимальне використання кешу

6. Як вибрати оптимальне k для k-way merge?
- 1) k = 2 завжди
- 2) k = M/B для максимального використання кешу
- 3) k = $\sqrt{n}$
- 4) Випадково

7. Чому cache-oblivious може програвати cache-aware?
- 1) Гірша асимптотика
- 2) Overhead рекурсії та гірші константи
- 3) Не використовує кеш
- 4) Застарілий підхід

8. Що таке cache-adaptive алгоритми?
- 1) Те саме що cache-oblivious
- 2) Динамічно адаптуються до поточного стану кешу
- 3) Працюють без кешу
- 4) Оптимізовані для GPU

9. Як профілювати cache-oblivious алгоритми?
- 1) Неможливо
- 2) Hardware performance counters для cache misses
- 3) Тільки за часом
- 4) За споживанням енергії

10. Коли НЕ варто використовувати cache-oblivious підхід?

- 1) Для матричних операцій
- 2) Коли розмір даних фіксований і відомі параметри системи
- 3) Для сортування
- 4) Завжди варто

Відповіді:

1. 2) Гарантує що підзадачі поміщаються в кеш
2. 2) Рекурсія створює підзадачі які відповідають кожному рівню кешу
3. 2) Балансує між обходом в глибину та ширину
4. 2) k-funnel структура мінімізує cache misses
5. 2) Занадто дрібний поділ створює overhead без користі
6. 2) Максимально використовує доступний кеш
7. 2) Універсальність коштує додаткових витрат
8. 2) Активно моніторять та адаптуються до кешу
9. 2) Прямі вимірювання cache misses найточніші
10. 2) Спеціалізоване рішення може бути ефективнішим

Завдання для самостійної роботи

1. Реалізуйте cache-oblivious алгоритм для LU-розкладу матриці.
2. Створіть cache-oblivious версію алгоритму Штрассена.
3. Розробіть cache-oblivious структуру даних для range queries.
4. Реалізуйте cache-oblivious FFT (швидке перетворення Фур'є).
5. Створіть cache-oblivious алгоритм для задачі найближчих сусідів.
6. Розробіть cache-oblivious B-дерево з оптимальними операціями.
7. Реалізуйте cache-oblivious алгоритм побудови суфіксного масиву.
8. Створіть бібліотеку cache-oblivious алгоритмів для лінійної алгебри.
9. Розробіть інструмент для автоматичного перетворення алгоритмів у cache-oblivious.
10. Дослідіть застосування cache-oblivious підходу для GPU алгоритмів.

Теми лабораторних робіт

1. Базові cache-oblivious алгоритми та їх аналіз.
2. Cache-oblivious структури даних для пошуку.

3. Порівняння cache-oblivious та cache-aware матричних алгоритмів.
4. Розробка cache-oblivious бібліотеки сортування.
5. Cache-oblivious алгоритми для обробки графів.
6. Адаптація класичних алгоритмів до cache-oblivious моделі.
7. Cache-oblivious алгоритми для стиснення даних.
8. Візуалізація та профілювання cache-oblivious алгоритмів.
9. Cache-oblivious паралельні алгоритми.
10. Застосування cache-oblivious підходу в реальних системах.

Розділ 15: Розділ 15: Вступ до алгоритмів машинного навчання

Досі ми розглядали алгоритми як чіткі, детерміновані інструкції, створені людиною для розв'язання конкретної задачі. У цьому розділі ми переходимо до нової парадигми: машинного навчання (ML), де алгоритми вчаться виконувати завдання самостійно, аналізуючи дані та знаходячи в них закономірності. Замість того, щоб програмувати правила вручну, ми надаємо комп'ютеру можливість "навчитися" їх на основі прикладів.

Цей розділ є вступом у світ, де комп'ютери можуть прогнозувати, класифікувати, групувати та перетворювати дані способами, які раніше були під силу лише людському інтелекту. Ми дослідимо фундаментальні концепції, що лежать в основі сучасного штучного інтелекту, і розглянемо ключові типи завдань та методи їх вирішення.

Ми почнемо з навчання з учителем, де моделі вчаться на даних із "правильними відповідями":

- Регресія дозволить нам прогнозувати неперервні значення, як-от ціну акцій чи температуру.

- Класифікація навчить моделі присвоювати об'єктам мітки — наприклад, розрізняти спам від звичайних листів.

Далі ми зануримось у навчання без учителя, де алгоритми шукають приховані структури в нерозмічених даних:

- Кластеризація допоможе нам знаходити природні групи та сегменти в даних.

- Зниження розмірності покаже, як можна спростити складні дані, зберігши найважливішу інформацію.

Нарешті, ми торкнемося нейронних мереж — потужного інструменту, що імітує роботу людського мозку і лежить в основі революції глибокого навчання (Deep Learning).

Цей розділ закладе фундамент для розуміння того, як працюють сучасні інтелектуальні системи, від рекомендаційних сервісів до розпізнавання образів.

Лінійна регресія

Теоретичні відомості

Лінійна регресія є фундаментальним статистичним методом для моделювання залежності між змінними. Цей метод був розроблений у XIX

столітті Френсісом Гальтоном та Карлом Пірсоном і став основою для багатьох сучасних методів машинного навчання та статистичного аналізу.

Лінійна регресія моделює зв'язок між залежною змінною y (яку також називають цільовою змінною або відгуком) та однією або кількома незалежними змінними $x_1, x_2, \ldots, x_p$ (які називають предикторами, ознаками або регресорами). Основне припущення полягає в тому, що цей зв'язок є лінійним, тобто зміна залежної змінної пропорційна зміні незалежних змінних.

Для простої лінійної регресії з однією незалежною змінною модель має вигляд:

$$y = \beta_0 + \beta_1 x + \epsilon$$

де: - y - залежна змінна - x - незалежна змінна - β_0 - вільний член (intercept), значення y при $x = 0$ - β_1 - коефіцієнт нахилу (slope), показує на скільки змінюється y при зміні x на одиницю - ϵ - випадкова похибка, яка враховує вплив неврахованих факторів

Для множинної лінійної регресії з p незалежними змінними:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p + \epsilon$$

У векторній формі це можна записати як:

$$\mathbf{y} = \mathbf{X\beta} + \boldsymbol{\epsilon}$$

де: - $\mathbf{y}$ - вектор спостережень розміру $n \times 1$ - $\mathbf{X}$ - матриця ознак розміру $n \times (p + 1)$ (включаючи стовпець одиниць для вільного члена) - $\boldsymbol{\beta}$ - вектор параметрів розміру $(p + 1) \times 1$ - $\boldsymbol{\epsilon}$ - вектор похибок розміру $n \times 1$

Метод найменших квадратів

Основним методом оцінки параметрів лінійної регресії є метод найменших квадратів (МНК). Ідея методу полягає в мінімізації суми квадратів відхилень спостережуваних значень від передбачених моделлю:

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_{i1} - \ldots - \beta_p x_{ip})^2$$

де $\hat{y}_i$ - передбачене значення для i-го спостереження.

Для знаходження оптимальних параметрів необхідно взяти часткові похідні RSS по кожному параметру та прирівняти їх до нуля. Це приводить до системи нормальних рівнянь:

$$\mathbf{X}^T\mathbf{X\beta} = \mathbf{X}^T\mathbf{y}$$

Розв’язок цієї системи дає оцінки параметрів:

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

Припущення лінійної регресії

Для коректного застосування методу найменших квадратів та отримання незміщених оцінок необхідно виконання наступних припущень:

1. Лінійність: Залежність між змінними є лінійною

2. Незалежність спостережень: Спостереження незалежні одне від одного

3. Гомоскедастичність: Дисперсія похибок постійна для всіх спостережень

4. Нормальність похибок: Похибки розподілені за нормальним законом з нульовим середнім

5. Відсутність мультиколінеарності: Незалежні змінні не сильно корельовані між собою

Розглянемо алгоритм побудови моделі лінійної регресії:

```
struct LinearRegressionData {
    X: matrix[n][p+1]     // матриця ознак з стовпцем одиниць
    y: vector[n]          // вектор цільових значень
    n: integer            // кількість спостережень
    p: integer            // кількість ознак (без вільного члена)
}

struct LinearRegressionModel {
    beta: vector[p+1]     // вектор коефіцієнтів
    residuals: vector[n]  // вектор залишків
    R_squared: float      // коефіцієнт детермінації
    sigma: float          // стандартна похибка регресії
}

function fitLinearRegression(data: LinearRegressionData): LinearRegre
ssionModel
    // Крок 1: Додаємо стовпець одиниць для вільного члена
    X = addInterceptColumn(data.X)

    // Крок 2: Обчислюємо X^T * X
    XtX = matrixMultiply(transpose(X), X)

    // Крок 3: Перевіряємо на вироджність
    if determinant(XtX) == 0 then
        throw Error("Матриця X^T*X вироджена - проблема мультиколінеа
рності")
```

```
    // Крок 4: Обчислюємо (X^T * X)^(-1)
    XtX_inv = matrixInverse(XtX)

    // Крок 5: Обчислюємо X^T * y
    Xty = matrixVectorMultiply(transpose(X), data.y)

    // Крок 6: Знаходимо коефіцієнти beta = (X^T * X)^(-1) * X^T * y
    beta = matrixVectorMultiply(XtX_inv, Xty)

    // Крок 7: Обчислюємо передбачені значення
    y_pred = matrixVectorMultiply(X, beta)

    // Крок 8: Обчислюємо залишки
    residuals = vectorSubtract(data.y, y_pred)

    // Крок 9: Обчислюємо метрики якості
    RSS = dotProduct(residuals, residuals)
    y_mean = mean(data.y)
    TSS = sum((data.y[i] - y_mean)^2 for i in 1..n)
    R_squared = 1 - RSS / TSS

    // Крок 10: Обчислюємо стандартну похибку
    sigma = sqrt(RSS / (data.n - data.p - 1))

    return LinearRegressionModel(beta, residuals, R_squared, sigma)

function addInterceptColumn(X: matrix): matrix
    // Додаємо стовпець одиниць як перший стовпець
    n = rows(X)
    p = cols(X)
    X_with_intercept = matrix[n][p+1]

    for i = 1 to n do
        X_with_intercept[i][1] = 1
        for j = 1 to p do
            X_with_intercept[i][j+1] = X[i][j]

    return X_with_intercept
```

Градієнтний спуск для лінійної регресії

Альтернативним методом знаходження параметрів є градієнтний спуск, який особливо корисний для великих датасетів:

```
function gradientDescentLinearRegression(data: LinearRegressionData,
                                         learning_rate: float,
```

```
                                        max_iterations: integer,
                                        tolerance: float): LinearRegr
essionModel
    // Ініціалізація параметрів
    beta = initializeRandomVector(data.p + 1)
    X = addInterceptColumn(data.X)
    n = data.n

    for iteration = 1 to max_iterations do
        // Обчислюємо передбачення
        y_pred = matrixVectorMultiply(X, beta)

        // Обчислюємо похибки
        errors = vectorSubtract(y_pred, data.y)

        // Обчислюємо градієнт функції втрат
        gradient = vectorScalarMultiply(
            matrixVectorMultiply(transpose(X), errors),
            2.0 / n
        )

        // Оновлюємо параметри
        beta_new = vectorSubtract(beta,
                                  vectorScalarMultiply(gradient, learn
ing_rate))

        // Перевіряємо збіжність
        if norm(vectorSubtract(beta_new, beta)) < tolerance then
            break

        beta = beta_new

    // Обчислюємо фінальні метрики
    y_pred = matrixVectorMultiply(X, beta)
    residuals = vectorSubtract(data.y, y_pred)

    return computeModelMetrics(beta, residuals, data)

function computeCostFunction(X: matrix, y: vector, beta: vector): flo
at
    // Функція втрат - середньоквадратична похибка
    n = length(y)
    y_pred = matrixVectorMultiply(X, beta)
    errors = vectorSubtract(y_pred, y)
```

```
    return dotProduct(errors, errors) / (2 * n)
```

Регуляризація в лінійній регресії

Для запобігання перенавчанню використовують регуляризовані версії лінійної регресії:

1. Ridge регресія (L2 регуляризація)

```
function ridgeRegression(data: LinearRegressionData, lambda: float):
LinearRegressionModel
    X = addInterceptColumn(data.X)

    // Створюємо матрицю регуляризації
    I = identityMatrix(data.p + 1)
    I[1][1] = 0  // не регуляризуємо вільний член

    // Ridge оцінка: beta = (X^T*X + lambda*I)^(-1) * X^T*y
    XtX = matrixMultiply(transpose(X), X)
    XtX_regularized = matrixAdd(XtX, scalarMatrixMultiply(lambda, I))

    XtX_inv = matrixInverse(XtX_regularized)
    Xty = matrixVectorMultiply(transpose(X), data.y)
    beta = matrixVectorMultiply(XtX_inv, Xty)

    return computeModelFromBeta(beta, X, data.y)
```

2. Lasso регресія (L1 регуляризація)

```
function lassoRegression(data: LinearRegressionData, lambda: float,
                         max_iterations: integer): LinearRegressionMod
el
    // Використовуємо coordinate descent
    X = standardizeFeatures(addInterceptColumn(data.X))
    beta = initializeZeroVector(data.p + 1)

    for iteration = 1 to max_iterations do
        beta_old = copy(beta)

        // Оновлюємо кожен коефіцієнт окремо
        for j = 1 to data.p + 1 do
            // Обчислюємо залишки без j-ої змінної
            r_j = computePartialResiduals(X, data.y, beta, j)

            // Обчислюємо кореляцію
            rho_j = dotProduct(X[:, j], r_j) / data.n

            // Soft thresholding
```

```
            if j == 1 then  // вільний член
                beta[j] = rho_j
            else
                beta[j] = softThreshold(rho_j, lambda)

        // Перевіряємо збіжність
        if norm(vectorSubtract(beta, beta_old)) < 1e-6 then
            break

    return computeModelFromBeta(beta, X, data.y)

function softThreshold(x: float, lambda: float): float
    if x > lambda then
        return x - lambda
    else if x < -lambda then
        return x + lambda
    else
        return 0
```

Діагностика моделі

Важливою частиною побудови регресійної моделі є її діагностика:

```
function diagnosticLinearRegression(model: LinearRegressionModel,
                                    X: matrix, y: vector): DiagnosticR
eport
    n = length(y)
    p = cols(X) - 1

    // Обчислюємо стандартизовані залишки
    standardized_residuals = []
    for i = 1 to n do
        h_ii = computeLeverage(X, i)
        std_error = model.sigma * sqrt(1 - h_ii)
        standardized_residuals[i] = model.residuals[i] / std_error

    // Тест на нормальність залишків (Shapiro-Wilk)
    normality_test = shapiroWilkTest(standardized_residuals)

    // Тест на гомоскедастичність (Breusch-Pagan)
    homoscedasticity_test = breuschPaganTest(model.residuals, X)

    // Обчислюємо VIF для мультиколінеарності
    vif_values = []
    for j = 2 to p + 1 do  // пропускаємо вільний член
        vif_values[j-1] = computeVIF(X, j)
```

```
    // Виявлення викидів
    outliers = detectOutliers(standardized_residuals, X)

    return DiagnosticReport(
        normality_test,
        homoscedasticity_test,
        vif_values,
        outliers
    )

function computeVIF(X: matrix, column: integer): float
    // Variance Inflation Factor
    X_j = X[:, column]
    X_others = removeColumn(X, column)

    // Регресія X_j на інші змінні
    model = fitLinearRegression(X_others, X_j)

    return 1 / (1 - model.R_squared)

function computeLeverage(X: matrix, i: integer): float
    // Hat matrix: H = X(X^T*X)^(-1)X^T
    // Leverage = діагональний елемент H
    XtX_inv = matrixInverse(matrixMultiply(transpose(X), X))
    x_i = X[i, :]

    return dotProduct(x_i, matrixVectorMultiply(XtX_inv, x_i))
```

Прогнозування та довірчі інтервали

```
function predict(model: LinearRegressionModel, X_new: matrix): Predic
tionResult
    // Додаємо стовпець одиниць
    X_new_with_intercept = addInterceptColumn(X_new)

    // Точкові прогнози
    y_pred = matrixVectorMultiply(X_new_with_intercept, model.beta)

    // Обчислюємо стандартні похибки прогнозів
    XtX_inv = model.XtX_inv  // збережене з етапу навчання
    se_pred = []

    for i = 1 to rows(X_new) do
        x_i = X_new_with_intercept[i, :]
        var_pred = model.sigma^2 * (1 + dotProduct(x_i,
                                    matrixVectorMultiply(XtX_inv, x_i)
```

```
))
        se_pred[i] = sqrt(var_pred)

    // Обчислюємо довірчі інтервали (95%)
    t_critical = tDistributionQuantile(0.975, model.df)

    lower_bound = []
    upper_bound = []

    for i = 1 to length(y_pred) do
        margin = t_critical * se_pred[i]
        lower_bound[i] = y_pred[i] - margin
        upper_bound[i] = y_pred[i] + margin

    return PredictionResult(y_pred, se_pred, lower_bound, upper_bound
)
```

Область застосування

Лінійна регресія широко застосовується в багатьох галузях:

- Економіка та фінанси: Прогнозування цін, аналіз попиту та пропозиції, оцінка ризиків, моделювання макроекономічних показників.

- Маркетинг: Аналіз ефективності реклами, прогнозування продажів, сегментація клієнтів, оптимізація ціноутворення.

- Медицина: Прогнозування захворювань, аналіз факторів ризику, дозування ліків, епідеміологічні дослідження.

- Інженерія: Калібрування датчиків, прогнозування надійності, оптимізація процесів, контроль якості.

- Соціальні науки: Аналіз соціальних явищ, прогнозування поведінки, оцінка впливу політик.

- Машинне навчання: Базовий метод для порівняння, feature engineering, ансамблеві методи.

Приклади застосування

Приклад 1: Проста лінійна регресія

Розглянемо прогнозування ціни будинку на основі його площі:

```
Дані (площа в м², ціна в тис. грн):
X = [50, 70, 90, 110, 130]
y = [1500, 2000, 2400, 2900, 3300]

Крок 1: Додаємо стовпець одиниць
```

```
X_with_intercept = [[1, 50],
                    [1, 70],
                    [1, 90],
                    [1, 110],
                    [1, 130]]

Крок 2: Обчислюємо X^T * X
X^T * X = [[5, 450],
           [450, 45500]]

Крок 3: Обчислюємо (X^T * X)^(-1)
det = 5 * 45500 - 450 * 450 = 25000
(X^T * X)^(-1) = [[1.82, -0.018],
                  [-0.018, 0.0002]]

Крок 4: Обчислюємо X^T * y
X^T * y = [12100, 1195000]

Крок 5: Знаходимо коефіцієнти
beta = (X^T * X)^(-1) * X^T * y = [418, 23.1]

Модель: ціна = 418 + 23.1 * площа

Інтерпретація:
- Базова ціна (при площі 0): 418 тис. грн
- Кожен м² додає 23.1 тис. грн до ціни

Прогноз для будинку 100 м²:
ціна = 418 + 23.1 * 100 = 2728 тис. грн
```

Приклад 2: Множинна лінійна регресія

Прогнозування споживання палива автомобілем на основі ваги та потужності:

```
struct CarData {
    weight: float    // вага в кг
    power: float     // потужність в к.с.
    fuel: float      // споживання л/100км
}

// Дані про 5 автомобілів
cars = [
    {1200, 90, 6.5},
    {1400, 110, 7.2},
    {1600, 130, 8.1},
    {1800, 150, 9.0},
```

```
    {2000, 170, 9.8}
]

// Формуємо матриці
X = [[1, 1200, 90],
     [1, 1400, 110],
     [1, 1600, 130],
     [1, 1800, 150],
     [1, 2000, 170]]

y = [6.5, 7.2, 8.1, 9.0, 9.8]

// Застосовуємо алгоритм
model = fitLinearRegression(X, y)

// Результат:
beta = [-1.5, 0.003, 0.025]

Модель: споживання = -1.5 + 0.003 * вага + 0.025 * потужність

R² = 0.997 (модель пояснює 99.7% варіації)

Інтерпретація:
- Кожні 100 кг ваги додають 0.3 л/100км
- Кожні 10 к.с. потужності додають 0.25 л/100км
```

Приклад 3: Регресія з регуляризацією

Порівняння звичайної та Ridge регресії при мультиколінеарності:

```
// Дані з сильною кореляцією між ознаками
X = [[1, 2],
     [2, 4.1],
     [3, 5.9],
     [4, 8.2],
     [5, 10.1]]

y = [5.2, 8.9, 12.1, 15.8, 19.2]

// Звичайна регресія
ordinary_model = fitLinearRegression(X, y)
// Результат нестабільний через мультиколінеарність
// beta може бути [1.5, 15.2, -5.1] з великими коефіцієнтами

// Ridge регресія з lambda = 1.0
ridge_model = ridgeRegression(X, y, lambda = 1.0)
// Результат стабільніший
```

```
// beta = [0.8, 2.1, 1.8] - менші, збалансовані коефіцієнти

// Порівняння на тестових даних
X_test = [[6, 12.1]]
ordinary_pred = 23.5 (може бути нестабільним)
ridge_pred = 22.8 (стабільніший прогноз)
```

Приклад 4: Градієнтний спуск

Покрокова ілюстрація градієнтного спуску:

```
// Проста задача: y = 2x + 1 + шум
X = [1, 2, 3, 4, 5]
y = [3.1, 4.9, 7.2, 8.8, 11.1]

// Параметри
learning_rate = 0.01
beta = [0, 0]  // початкові значення

Ітерація 1:
- Прогнози: [0, 0, 0, 0, 0]
- Похибки: [-3.1, -4.9, -7.2, -8.8, -11.1]
- Градієнт: [-7.02, -21.7]
- Оновлення: beta = [0.07, 0.22]

Ітерація 2:
- Прогнози: [0.29, 0.51, 0.73, 0.95, 1.17]
- Похибки: [-2.81, -4.39, -6.47, -7.85, -9.93]
- Градієнт: [-6.29, -19.37]
- Оновлення: beta = [0.13, 0.41]

...

Ітерація 100:
- beta = [0.98, 1.99]
- Близько до справжніх значень [1, 2]
```

Приклад 5: Комплексний аналіз з діагностикою

Повний pipeline аналізу даних про зарплати:

```
struct SalaryAnalysis {
    experience: float   // роки досвіду
    education: float    // роки освіти
    age: float         // вік
    salary: float      // зарплата в тис. грн/місяць
}

function completeSalaryAnalysis(data: array of SalaryAnalysis)
```

```
    // Етап 1: Підготовка даних
    X, y = prepareData(data)
    X_train, X_test, y_train, y_test = trainTestSplit(X, y, test_size
= 0.2)

    // Етап 2: Побудова базової моделі
    base_model = fitLinearRegression(X_train, y_train)

    // Етап 3: Діагностика
    diagnostics = diagnosticLinearRegression(base_model, X_train, y_t
rain)

    // Виявлено мультиколінеарність між віком та досвідом
    // VIF для віку = 8.5, для досвіду = 7.2

    // Етап 4: Виправлення - видаляємо вік
    X_reduced = removeColumn(X, "age")
    improved_model = fitLinearRegression(X_reduced_train, y_train)

    // Етап 5: Регуляризація для стабільності
    ridge_model = ridgeRegression(X_train, y_train, lambda = 0.5)

    // Етап 6: Порівняння моделей на тестовій вибірці
    base_rmse = RMSE(base_model.predict(X_test), y_test)     // 3.2
    improved_rmse = RMSE(improved_model.predict(X_test), y_test) // 3
.1
    ridge_rmse = RMSE(ridge_model.predict(X_test), y_test)   // 3.0

    // Етап 7: Фінальна модель з інтерпретацією
    final_model = ridge_model

    // Коефіцієнти:
    // - Кожен рік досвіду додає 1.8 тис. грн
    // - Кожен рік освіти додає 2.3 тис. грн
    // - Базова зарплата: 15 тис. грн

    // Етап 8: Довірчі інтервали для нового працівника
    new_employee = {experience: 5, education: 16}
    prediction = predict(final_model, new_employee)
    // Прогноз: 36.4 тис. грн
    // 95% довірчий інтервал: [33.2, 39.6] тис. грн
```

Переваги та недоліки

Переваги

- Простота та інтерпретованість: Коефіцієнти мають чітку інтерпретацію - вплив кожної змінної на результат.

- Швидкість обчислень: Аналітичний розв’язок через нормальні рівняння або ефективні ітеративні методи.

- Статистична теорія: Розвинена теорія для тестування гіпотез, побудови довірчих інтервалів.

- Не потребує налаштування: Базова версія не має гіперпараметрів (крім регуляризованих версій).

- Екстраполяція: Може робити прогнози за межами діапазону навчальних даних.

- Базовий метод: Служить основою для порівняння з складнішими моделями.

Недоліки

- Лінійність: Не може моделювати нелінійні залежності без додаткової обробки ознак.

- Чутливість до викидів: Метод найменших квадратів сильно реагує на аномальні значення.

- Припущення про розподіл: Для статистичних висновків потрібна нормальність похибок.

- Мультиколінеарність: Проблеми при сильній кореляції між незалежними змінними.

- Перенавчання: При великій кількості ознак відносно спостережень.

- Гомоскедастичність: Припущення про постійну дисперсію часто порушується на практиці.

Контрольні запитання

1. Що таке лінійна регресія та які основні припущення методу найменших квадратів?

2. Як інтерпретувати коефіцієнти в моделі лінійної регресії?

3. Що таке коефіцієнт детермінації R^2 та як його обчислити?

4. Яка різниця між простою та множинною лінійною регресією?

5. Що таке мультиколінеарність та як її виявити?

6. Як працює градієнтний спуск для лінійної регресії?

7. Що таке регуляризація та навіщо вона потрібна?

8. Яка різниця між Ridge та Lasso регресією?

9. Як побудувати довірчі інтервали для прогнозів?

10. Які методи діагностики використовуються для перевірки адекватності моделі?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження методів лінійної регресії

2. Мета

Вивчити теоретичні основи лінійної регресії, реалізувати різні методи оцінки параметрів, навчитися діагностувати та покращувати регресійні моделі, порівняти різні підходи до регуляризації.

3. Завдання

Реалізувати метод найменших квадратів та градієнтний спуск, додати підтримку Ridge та Lasso регресії, створити систему діагностики моделі, провести експерименти на реальних даних, візуалізувати результати.

4. Опис

Лабораторна робота включає повний цикл побудови регресійної моделі: від підготовки даних до інтерпретації результатів. Студенти реалізують основні алгоритми, методи діагностики та візуалізації.

5. Порядок виконання

- Реалізувати базовий метод найменших квадратів
- Додати обчислення статистичних метрик (R^2, RMSE, MAE)
- Реалізувати градієнтний спуск з різними learning rate
- Додати Ridge та Lasso регресію
- Створити функції для діагностики (VIF, тести на нормальність)
- Реалізувати побудову довірчих інтервалів
- Візуалізувати залишки та Q-Q plots
- Провести cross-validation для вибору гіперпараметрів
- Порівняти методи на різних датасетах
- Створити інтерактивний інтерфейс для аналізу

6. Результат

Повнофункціональна бібліотека для лінійної регресії з підтримкою різних методів оцінки, система автоматичної діагностики моделей, візуалізація результатів, звіт з порівнянням методів на реальних даних.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас LinearRegression з методами fit, predict

- Реалізація градієнтного спуску

- Ridge та Lasso регресія

- Функції діагностики та візуалізації

- Система cross-validation

2. Продемонструвати роботу додатку:

- Побудова моделі на навчальних даних

- Візуалізація залежностей та залишків

- Порівняння різних методів регуляризації

- Діагностика порушень припущень

- Інтерпретація результатів

3. Бути готовим відповісти на питання:

- Коли використовувати Ridge vs Lasso?

- Як вибрати оптимальний параметр регуляризації?

- Що робити при порушенні припущень?

- Як інтерпретувати коефіцієнти при мультиколінеарності?

Тести

1. Метод найменших квадратів мінімізує:

- 1) Суму абсолютних відхилень

- 2) Суму квадратів відхилень

- 3) Максимальне відхилення

- 4) Кількість помилок

2. Коефіцієнт детермінації R^2 показує:

- 1) Кореляцію між змінними

- 2) Частку поясненої дисперсії
- 3) Стандартну похибку
- 4) Значущість моделі

3. При мультиколінеарності:
- 1) Модель не працює
- 2) Коефіцієнти нестабільні
- 3) $R^2 = 0$
- 4) Прогнози неможливі

4. Ridge регресія додає до функції втрат:
- 1) L1 норму коефіцієнтів
- 2) L2 норму коефіцієнтів
- 3) Кількість ненульових коефіцієнтів
- 4) Максимальний коефіцієнт

5. Градієнтний спуск збігається до:
- 1) Глобального мінімуму для опуклих функцій
- 2) Випадкової точки
- 3) Початкового значення
- 4) Нуля

Відповіді:

1-2,

2-2,

3-2,

4-2,

5-1

Тести підвищеної складності

1. Чому матриця X^T*X може бути виродженою?
- 1) Мало даних
- 2) Лінійна залежність між стовпцями X
- 3) Великі значення в X
- 4) Помилки в даних

2. Що таке leverage в контексті регресії?

- 1) Коефіцієнт моделі
- 2) Вплив спостереження на власне передбачення
- 3) Похибка прогнозу
- 4) Тип регуляризації

3. Коли Lasso може обнулити коефіцієнти?

- 1) Ніколи
- 2) При великому параметрі регуляризації
- 3) При малому параметрі регуляризації
- 4) Випадково

4. Що означає гетероскедастичність?

- 1) Нелінійна залежність
- 2) Непостійна дисперсія похибок
- 3) Ненормальність похибок
- 4) Мультиколінеарність

5. Яка складність обчислення МНК через нормальні рівняння?

- 1) O(n)
- 2) O(np)
- 3) $O(np^2 + p^3)$
- 4) $O(n^2p)$

6. Чому використовують стандартизацію ознак?

- 1) Для красоти
- 2) Щоб коефіцієнти були порівнянними
- 3) Це обов’язково
- 4) Для швидкості

7. Що таке умовне число матриці X^T*X?

- 1) Кількість рядків
- 2) Міра близькості до виродженості
- 3) Детермінант
- 4) Ранг

8. Як elastic net поєднує Ridge та Lasso?

- 1) Використовує обидва по черзі
- 2) Лінійна комбінація L1 та L2 штрафів
- 3) Вибирає кращий
- 4) Не поєднує

9. Що таке influential points?

- 1) Точки з великими значеннями
- 2) Точки що сильно впливають на модель
- 3) Центральні точки
- 4) Початкові точки

10. Коли використовувати робастна (інтервальна) регресія?

- 1) Завжди
- 2) При наявності викидів
- 3) Для великих даних
- 4) Для малих даних

Відповіді:

1. 2) Лінійно залежні стовпці роблять матрицю виродженою
2. 2) Діагональні елементи hat matrix показують self-influence
3. 2) L1 штраф може обнулити коефіцієнти при достатній силі
4. 2) Змінна дисперсія порушує припущення МНК
5. 3) Множення матриць $O(np^2)$ + обернення $O(p^3)$
6. 2) Робить коефіцієнти незалежними від масштабу
7. 2) Відношення max/min власних значень
8. 2) $\alpha L1 + (1-\alpha)$L2 для балансу
9. 2) Cook's distance вимірює вплив на всі прогнози
10. 2) Методи як Huber regression стійкі до викидів

Завдання для самостійної роботи

1. Реалізуйте поліноміальну регресію через розширення ознак.
2. Створіть автоматичний вибір ознак через forward/backward selection.
3. Розробіть bootstrap для оцінки невизначеності коефіцієнтів.

4. Реалізуйте квантильну регресію для прогнозування медіани.

5. Створіть weighted least squares для гетероскедастичних даних.

6. Розробіть online learning версію з оновленням параметрів.

7. Реалізуйте partial least squares (PLS) регресію.

8. Створіть інструмент для автоматичної діагностики моделі.

9. Розробіть метод для роботи з пропущеними даними.

10. Дослідіть байєсівську лінійну регресію.

Теми лабораторних робіт

1. Базова реалізація лінійної регресії та візуалізація.

2. Порівняння методів оцінки параметрів (МНК, градієнтний спуск, стохастичний градієнт).

3. Дослідження методів регуляризації (Ridge, Lasso, Elastic Net).

4. Діагностика регресійних моделей та виявлення порушень.

5. Робота з категоріальними змінними та взаємодіями.

6. Time series регресія з автокореляцією.

7. Множинні порівняння та корекція p-values.

8. Створення пакету для автоматичного ML з регресією.

9. Порівняння лінійної регресії з нелінійними методами.

10. Застосування регресії для реальної бізнес-задачі.

Логістична регресія

Теоретичні відомості

Логістична регресія є фундаментальним статистичним методом для вирішення задач бінарної та багатокласової класифікації. Незважаючи на назву “регресія”, цей метод використовується саме для класифікації, оскільки передбачає ймовірність належності об’єкта до певного класу. Метод був розроблений статистиком Девідом Коксом у 1958 році та став одним з найпоширеніших алгоритмів машинного навчання завдяки своїй простоті, інтерпретованості та ефективності.

На відміну від лінійної регресії, яка передбачає неперервні значення, логістична регресія моделює ймовірність належності спостереження до певного класу. Для бінарної класифікації ми маємо два класи, які зазвичай

позначають як 0 та 1, де 1 означає “позитивний” клас (наявність певної властивості), а 0 - “негативний” клас (відсутність властивості).

Основна проблема прямого застосування лінійної регресії для класифікації полягає в тому, що лінійна комбінація ознак може давати будь-які значення від $-\infty$ до $+\infty$, тоді як ймовірність повинна лежати в діапазоні [0,1]. Логістична регресія вирішує цю проблему, застосовуючи логістичну (сигмоїдну) функцію до лінійної комбінації ознак.

Логістична функція (сигмоїда) визначається як:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

де z - будь-яке дійсне число. Ця функція має важливі властивості: - Приймає значення від 0 до 1 - Монотонно зростає - Має S-подібну форму - $\sigma(0) = 0.5$ - $\lim_{z \to +\infty} \sigma(z) = 1$ - $\lim_{z \to -\infty} \sigma(z) = 0$

Для логістичної регресії ми моделюємо ймовірність належності до класу 1 як:

$$P(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x} + b)}}$$

де: - $\mathbf{x}$ - вектор ознак розміру d - $\mathbf{w}$ - вектор ваг (параметрів) розміру d - b - зсув (bias) - $\mathbf{w}^T\mathbf{x}$ - скалярний добуток векторів

Відповідно, ймовірність належності до класу 0:

$$P(y = 0|\mathbf{x}) = 1 - P(y = 1|\mathbf{x}) = \frac{e^{-(\mathbf{w}^T\mathbf{x} + b)}}{1 + e^{-(\mathbf{w}^T\mathbf{x} + b)}}$$

Функція втрат

Для навчання моделі необхідно визначити функцію втрат, яку будемо мінімізувати. У логістичній регресії використовується функція втрат на основі принципу максимальної правдоподібності. Для одного спостереження функція правдоподібності:

$$L(y_i|\mathbf{x}_i, \mathbf{w}, b) = p_i^{y_i}(1 - p_i)^{1 - y_i}$$

де $p_i = P(y_i = 1|\mathbf{x}_i)$.

Беручи логарифм та змінюючи знак, отримуємо бінарну крос-ентропію (binary cross-entropy):

$$\mathcal{L}(y_i, \hat{y}_i) = -[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Для всього набору даних з n спостережень:

$$J(\mathbf{w}, b) = -\frac{1}{n}\sum_{i=1}^{n}[\, y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

Градієнти функції втрат

Для оптимізації параметрів методом градієнтного спуску необхідно обчислити часткові похідні функції втрат. Завдяки властивостям логістичної функції, градієнти мають елегантну форму:

$$\frac{\partial J}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)x_{ij}$$

$$\frac{\partial J}{\partial b} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)$$

де $\hat{y}_i = \sigma(\mathbf{w}^T\mathbf{x}_i + b)$ - передбачена ймовірність.

Розглянемо детальний алгоритм навчання моделі логістичної регресії методом градієнтного спуску:

```
struct LogisticRegressionData {
    X: matrix[n][d]     // матриця ознак
    y: vector[n]        // вектор міток класів (0 або 1)
    n: integer          // кількість спостережень
    d: integer          // кількість ознак
}

struct LogisticRegressionModel {
    w: vector[d]        // вектор ваг
    b: float            // зсув
    loss_history: array // історія функції втрат
}

function sigmoid(z: float): float
    // Обчислення сигмоїдної функції
    // Для численної стабільності використовуємо різні формули
    if z >= 0 then
        exp_neg_z = exp(-z)
        return 1 / (1 + exp_neg_z)
    else
        exp_z = exp(z)
        return exp_z / (1 + exp_z)

function logisticRegressionGradientDescent(data: LogisticRegressionDa
ta,
```

```
                                                learning_rate: float,
                                                max_iterations: integer,
                                                tolerance: float): Logistic
RegressionModel
    // Ініціалізація параметрів
    w = initializeZeroVector(data.d)
    b = 0.0
    loss_history = []

    for iteration = 1 to max_iterations do
        // Крок 1: Обчислення лінійної комбінації
        z = matrixVectorMultiply(data.X, w) + b

        // Крок 2: Застосування сигмоїдної функції
        y_pred = []
        for i = 1 to data.n do
            y_pred[i] = sigmoid(z[i])

        // Крок 3: Обчислення функції втрат
        loss = 0.0
        for i = 1 to data.n do
            // Додаємо малу константу для численної стабільності
            epsilon = 1e-15
            y_pred_clipped = clip(y_pred[i], epsilon, 1 - epsilon)

            loss += -(data.y[i] * log(y_pred_clipped) +
                     (1 - data.y[i]) * log(1 - y_pred_clipped))

        loss = loss / data.n
        loss_history.append(loss)

        // Крок 4: Обчислення градієнтів
        errors = vectorSubtract(y_pred, data.y)

        // Градієнт по w
        grad_w = vectorScalarMultiply(
            matrixVectorMultiply(transpose(data.X), errors),
            1.0 / data.n
        )

        // Градієнт по b
        grad_b = sum(errors) / data.n

        // Крок 5: Оновлення параметрів
        w_new = vectorSubtract(w, vectorScalarMultiply(grad_w, learni
```

```
ng_rate))
        b_new = b - learning_rate * grad_b

        // Крок 6: Перевірка збіжності
        if norm(vectorSubtract(w_new, w)) < tolerance and
           abs(b_new - b) < tolerance then
            break

        w = w_new
        b = b_new

    return LogisticRegressionModel(w, b, loss_history)
```

Стохастичний градієнтний спуск

Для великих наборів даних ефективнішим є стохастичний градієнтний спуск (SGD):

```
function logisticRegressionSGD(data: LogisticRegressionData,
                               learning_rate: float,
                               epochs: integer,
                               batch_size: integer): LogisticRegressio
nModel
    // Ініціалізація параметрів
    w = initializeRandomVector(data.d, mean=0, std=0.01)
    b = 0.0
    loss_history = []

    for epoch = 1 to epochs do
        // Перемішування даних
        indices = shuffle(range(1, data.n))

        epoch_loss = 0.0
        num_batches = ceil(data.n / batch_size)

        for batch = 1 to num_batches do
            // Вибір batch
            start_idx = (batch - 1) * batch_size + 1
            end_idx = min(batch * batch_size, data.n)
            batch_indices = indices[start_idx:end_idx]

            X_batch = data.X[batch_indices, :]
            y_batch = data.y[batch_indices]
            batch_n = length(batch_indices)

            // Forward pass
            z_batch = matrixVectorMultiply(X_batch, w) + b
```

```
            y_pred_batch = applySigmoid(z_batch)

            // Обчислення втрат для batch
            batch_loss = computeBinaryCrossEntropy(y_batch, y_pred_ba
tch)
            epoch_loss += batch_loss * batch_n

            // Backward pass
            errors = vectorSubtract(y_pred_batch, y_batch)

            grad_w = vectorScalarMultiply(
                matrixVectorMultiply(transpose(X_batch), errors),
                1.0 / batch_n
            )
            grad_b = sum(errors) / batch_n

            // Оновлення параметрів
            w = vectorSubtract(w, vectorScalarMultiply(grad_w, learni
ng_rate))
            b = b - learning_rate * grad_b

        // Середня втрата за епоху
        epoch_loss = epoch_loss / data.n
        loss_history.append(epoch_loss)

        // Адаптивний learning rate (опціонально)
        if epoch > 10 and loss_history[epoch] > loss_history[epoch-1]
then
            learning_rate = learning_rate * 0.9

    return LogisticRegressionModel(w, b, loss_history)
```

Регуляризація

Для запобігання перенавчанню застосовують L2 (Ridge) або L1 (Lasso) регуляризацію:

```
function logisticRegressionWithRegularization(data: LogisticRegressio
nData,
                                              learning_rate: float,
                                              lambda_reg: float,
                                              reg_type: string,
                                              max_iterations: integer)
: LogisticRegressionModel
    w = initializeZeroVector(data.d)
    b = 0.0
```

```
    for iteration = 1 to max_iterations do
        // Forward pass
        z = matrixVectorMultiply(data.X, w) + b
        y_pred = applySigmoid(z)

        // Обчислення втрат з регуляризацією
        ce_loss = computeBinaryCrossEntropy(data.y, y_pred)

        if reg_type == "L2" then
            reg_loss = lambda_reg * dotProduct(w, w) / 2
        else if reg_type == "L1" then
            reg_loss = lambda_reg * sum(abs(w[j]) for j in 1..d)
        else
            reg_loss = 0

        total_loss = ce_loss + reg_loss

        // Градієнти
        errors = vectorSubtract(y_pred, data.y)
        grad_w = vectorScalarMultiply(
            matrixVectorMultiply(transpose(data.X), errors),
            1.0 / data.n
        )

        // Додаємо регуляризаційний градієнт
        if reg_type == "L2" then
            grad_w = vectorAdd(grad_w, vectorScalarMultiply(w, lambda
_reg))
        else if reg_type == "L1" then
            for j = 1 to data.d do
                grad_w[j] += lambda_reg * sign(w[j])

        grad_b = sum(errors) / data.n

        // Оновлення параметрів
        w = vectorSubtract(w, vectorScalarMultiply(grad_w, learning_r
ate))
        b = b - learning_rate * grad_b

    return LogisticRegressionModel(w, b)
```

Багатокласова логістична регресія

Для задач з більше ніж двома класами використовують softmax функцію:

```
function softmax(z: vector[K]): vector[K]
    // Числено стабільна версія softmax
```

```
    z_max = max(z)
    z_stable = vectorSubtract(z, z_max)

    exp_z = []
    for k = 1 to K do
        exp_z[k] = exp(z_stable[k])

    sum_exp = sum(exp_z)

    probabilities = []
    for k = 1 to K do
        probabilities[k] = exp_z[k] / sum_exp

    return probabilities

function multiclassLogisticRegression(data: MulticlassData,
                                      learning_rate: float,
                                      max_iterations: integer): Multic
lassModel
    // data.y містить мітки класів від 0 до K-1
    K = max(data.y) + 1  // кількість класів
    d = data.d           // кількість ознак

    // Параметри: матриця ваг W[K][d] та вектор зсувів b[K]
    W = initializeMatrix(K, d, mean=0, std=0.01)
    b = initializeZeroVector(K)

    for iteration = 1 to max_iterations do
        // Forward pass
        logits = matrixMultiply(data.X, transpose(W)) + b  // [n][K]

        // Застосування softmax до кожного рядка
        probabilities = []
        for i = 1 to data.n do
            probabilities[i] = softmax(logits[i])

        // Обчислення крос-ентропійної втрати
        loss = 0.0
        for i = 1 to data.n do
            true_class = data.y[i]
            loss -= log(probabilities[i][true_class] + 1e-15)
        loss = loss / data.n

        // Backward pass
        // Обчислення градієнтів
```

```
        grad_W = initializeMatrix(K, d, 0)
        grad_b = initializeZeroVector(K)

        for i = 1 to data.n do
            // One-hot encoding істинного класу
            y_true = initializeZeroVector(K)
            y_true[data.y[i]] = 1

            // Похибка для i-го прикладу
            error = vectorSubtract(probabilities[i], y_true)

            // Накопичення градієнтів
            for k = 1 to K do
                for j = 1 to d do
                    grad_W[k][j] += error[k] * data.X[i][j] / data.n
                grad_b[k] += error[k] / data.n

        // Оновлення параметрів
        W = matrixSubtract(W, scalarMatrixMultiply(learning_rate, gra
d_W))
        b = vectorSubtract(b, vectorScalarMultiply(grad_b, learning_r
ate))

    return MulticlassModel(W, b)
```

Передбачення та оцінка якості

```
function predict(model: LogisticRegressionModel, X: matrix): array
    n = rows(X)
    predictions = []

    for i = 1 to n do
        z = dotProduct(model.w, X[i]) + model.b
        prob = sigmoid(z)

        // Класифікація з порогом 0.5
        if prob >= 0.5 then
            predictions[i] = 1
        else
            predictions[i] = 0

    return predictions

function predictProbabilities(model: LogisticRegressionModel, X: matr
ix): array
    n = rows(X)
```

```
    probabilities = []

    for i = 1 to n do
        z = dotProduct(model.w, X[i]) + model.b
        probabilities[i] = sigmoid(z)

    return probabilities

function evaluateModel(model: LogisticRegressionModel, X_test: matrix
, y_test: vector): Metrics
    y_pred = predict(model, X_test)
    y_prob = predictProbabilities(model, X_test)

    // Обчислення метрик
    TP = 0  // True Positives
    TN = 0  // True Negatives
    FP = 0  // False Positives
    FN = 0  // False Negatives

    for i = 1 to length(y_test) do
        if y_test[i] == 1 and y_pred[i] == 1 then
            TP += 1
        else if y_test[i] == 0 and y_pred[i] == 0 then
            TN += 1
        else if y_test[i] == 0 and y_pred[i] == 1 then
            FP += 1
        else if y_test[i] == 1 and y_pred[i] == 0 then
            FN += 1

    accuracy = (TP + TN) / (TP + TN + FP + FN)
    precision = TP / (TP + FP) if (TP + FP) > 0 else 0
    recall = TP / (TP + FN) if (TP + FN) > 0 else 0
    f1_score = 2 * precision * recall / (precision + recall) if (prec
ision + recall) > 0 else 0

    // AUC-ROC
    auc = computeAUC(y_test, y_prob)

    return Metrics(accuracy, precision, recall, f1_score, auc)
```

Методи оптимізації другого порядку

Для швидшої збіжності можна використовувати метод Ньютона:

```
function logisticRegressionNewton(data: LogisticRegressionData,
                                  max_iterations: integer,
                                  tolerance: float): LogisticRegressio
```

```
nModel
    // Розширюємо матрицю X стовпцем одиниць для зсуву
    X_extended = addOnesColumn(data.X)
    theta = initializeZeroVector(data.d + 1)

    for iteration = 1 to max_iterations do
        // Обчислення ймовірностей
        z = matrixVectorMultiply(X_extended, theta)
        p = applySigmoid(z)

        // Градієнт
        errors = vectorSubtract(p, data.y)
        gradient = vectorScalarMultiply(
            matrixVectorMultiply(transpose(X_extended), errors),
            1.0 / data.n
        )

        // Матриця Гессе
        // H = (1/n) * X^T * diag(p * (1-p)) * X
        diagonal_weights = []
        for i = 1 to data.n do
            diagonal_weights[i] = p[i] * (1 - p[i])

        weighted_X = []
        for i = 1 to data.n do
            weighted_X[i] = vectorScalarMultiply(X_extended[i],
                                                 sqrt(diagonal_weights[
i]))

        hessian = matrixMultiply(transpose(weighted_X), weighted_X)
        hessian = scalarMatrixMultiply(1.0 / data.n, hessian)

        // Додаємо регуляризацію для стабільності
        lambda_reg = 0.01
        I = identityMatrix(data.d + 1)
        hessian = matrixAdd(hessian, scalarMatrixMultiply(lambda_reg,
I))

        // Напрямок Ньютона
        newton_direction = solveLinearSystem(hessian, gradient)

        // Лінійний пошук для вибору кроку
        step_size = lineSearchBacktracking(data, theta, gradient,
                                           newton_direction)
```

```
        // Оновлення параметрів
        theta_new = vectorSubtract(theta,
                                   vectorScalarMultiply(newton_directi
on,
                                                        step_size))

        // Перевірка збіжності
        if norm(vectorSubtract(theta_new, theta)) < tolerance then
            break

        theta = theta_new

    // Розділення параметрів
    w = theta[1:data.d]
    b = theta[data.d + 1]

    return LogisticRegressionModel(w, b)
```

Область застосування

Логістична регресія широко застосовується в багатьох галузях:

- Медицина: Діагностика захворювань, прогнозування ризиків, аналіз виживаності пацієнтів.

- Фінанси: Кредитний скоринг, виявлення шахрайства, прогнозування дефолтів.

- Маркетинг: Прогнозування відтоку клієнтів, таргетована реклама, аналіз конверсій.

- Обробка природної мови: Класифікація тексту, аналіз тональності, фільтрація спаму.

- Комп'ютерний зір: Розпізнавання об'єктів, класифікація зображень.

- Біоінформатика: Класифікація генів, прогнозування структури білків.

Приклади застосування

Приклад 1: Бінарна класифікація - прогнозування покупки

Розглянемо задачу прогнозування, чи здійснить клієнт покупку на основі часу проведеного на сайті та кількості переглянутих товарів:

```
Дані:
time_on_site (хв) | items_viewed | purchased
       2.5        |      3       |     0
       5.1        |      8       |     1
       1.2        |      2       |     0
```

```
       4.8        |      7       |     1
       3.3        |      5       |     0
       6.2        |     10       |     1
```

Нормалізуємо дані:
```
X = [[-0.71, -0.87],
     [ 0.62,  0.87],
     [-1.42, -1.30],
     [ 0.41,  0.65],
     [-0.21, -0.22],
     [ 1.32,  1.52]]

y = [0, 1, 0, 1, 0, 1]
```

Ініціалізація: w = [0, 0], b = 0, learning_rate = 0.5

Ітерація 1:
```
z = Xw + b = [0, 0, 0, 0, 0, 0]
y_pred = sigmoid(z) = [0.5, 0.5, 0.5, 0.5, 0.5, 0.5]

Втрати = -1/6 * sum(y*log(0.5) + (1-y)*log(0.5)) = 0.693
```

Градієнти:
```
errors = [0.5, -0.5, 0.5, -0.5, 0.5, -0.5]
grad_w = 1/6 * X^T * errors = [-0.137, -0.151]
grad_b = 1/6 * sum(errors) = 0
```

Оновлення:
```
w = [0, 0] - 0.5 * [-0.137, -0.151] = [0.069, 0.076]
b = 0
```

Ітерація 2:
```
z = [0.018, 0.120, -0.020, 0.078, 0.002, 0.207]
y_pred = [0.504, 0.530, 0.495, 0.519, 0.501, 0.552]

Втрати = 0.688 (покращення!)

...продовження ітерацій...

Фінальна модель після 100 ітерацій:
w = [1.82, 2.15], b = 0.12
```

Інтерпретація:
- Кожна додаткова хвилина на сайті збільшує log-odds покупки на 1.82
- Кожен переглянутий товар збільшує log-odds на 2.15

Приклад 2: Медична діагностика

Прогнозування наявності діабету на основі медичних показників:

```
struct MedicalData {
    glucose: float      // рівень глюкози
    bmi: float          // індекс маси тіла
    age: float          // вік
    diabetes: integer  // 0 - немає, 1 - є
}

// Навчальні дані (спрощено)
train_data = [
    {120, 25.5, 35, 0},
    {165, 31.2, 52, 1},
    {100, 23.1, 28, 0},
    {180, 35.8, 61, 1},
    {110, 26.8, 42, 0}
]

// Модель з регуляризацією
model = logisticRegressionWithRegularization(
    train_data,
    learning_rate = 0.01,
    lambda_reg = 0.1,
    reg_type = "L2",
    max_iterations = 1000
)

// Результати:
// w = [0.082, 0.125, 0.048] (після нормалізації)
// b = -5.2

// Прогноз для нового пацієнта
new_patient = {glucose: 140, bmi: 29.0, age: 45}
probability = predictProbability(model, normalize(new_patient))
// probability = 0.73

// Інтерпретація: 73% ймовірність діабету
// Рекомендація: додаткові обстеження
```

Приклад 3: Багатокласова класифікація - розпізнавання цифр

Спрощений приклад розпізнавання рукописних цифр 0, 1, 2:

```
// Ознаки: спрощені характеристики зображення
// x1 - кількість вертикальних ліній
// x2 - кількість горизонтальних ліній
```

```
// x3 - наявність кола

Дані:
X = [[0, 1, 1],  // цифра 0
     [1, 0, 0],  // цифра 1
     [0, 2, 0],  // цифра 2
     [0, 1, 1],  // цифра 0
     [1, 0, 0],  // цифра 1
     [1, 2, 0]]  // цифра 2

y = [0, 1, 2, 0, 1, 2]

// Параметри після навчання:
W = [[-0.5,  0.8,  2.1],   // ваги для класу 0
     [ 2.3, -1.2, -0.9],   // ваги для класу 1
     [ 0.1,  2.5, -1.1]]   // ваги для класу 2

b = [-0.2, 0.1, 0.3]

// Прогноз для нового зображення
new_image = [1, 1, 0]  // можливо цифра 2

logits = Wx + b = [0.8, 0.2, 2.4]
probabilities = softmax(logits) = [0.21, 0.11, 0.68]

// Передбачення: клас 2 з ймовірністю 68%
```

Приклад 4: Аналіз тексту - фільтрація спаму

Класифікація електронних листів:

```
// Ознаки на основі частоти слів
features = {
    "contains_free": binary,
    "contains_money": binary,
    "contains_click": binary,
    "exclamation_count": integer,
    "length": integer
}

// Приклад даних
email_data = [
    {1, 1, 1, 5, 150, spam=1},  // "FREE MONEY! Click now!!!"
    {0, 0, 0, 0, 200, spam=0},  // звичайний лист
    {1, 0, 1, 3, 100, spam=1},  // "Free offer! Click here!"
    {0, 1, 0, 1, 180, spam=0},  // "Money transfer completed."
]
```

```
// Навчання моделі з різними порогами
model = trainLogisticRegression(email_data)

// Аналіз важливості ознак (ваги моделі):
// w[contains_free] = 2.8    - сильний індикатор спаму
// w[contains_money] = 0.5   - слабкий індикатор
// w[contains_click] = 2.1   - сильний індикатор
// w[exclamation_count] = 0.8 - помірний індикатор
// w[length] = -0.01         - довші листи рідше спам

// ROC аналіз для вибору порогу
thresholds = [0.3, 0.5, 0.7, 0.9]
for threshold in thresholds:
    metrics = evaluateWithThreshold(model, test_data, threshold)
    // threshold=0.3: high recall (90%), low precision (60%)
    // threshold=0.5: balanced (75% recall, 80% precision)
    // threshold=0.7: low recall (50%), high precision (95%)
```

Приклад 5: A/B тестування в маркетингу

Аналіз ефективності нового дизайну сайту:

```
struct UserInteraction {
    design_version: string    // "A" або "B"
    time_on_page: float      // секунди
    scroll_depth: float      // відсоток сторінки
    clicks: integer          // кількість кліків
    converted: integer       // 1 якщо купив, 0 якщо ні
}

// Підготовка даних для логістичної регресії
function prepareABTestData(interactions: array of UserInteraction)
    X = []
    y = []

    for interaction in interactions do
        features = [
            interaction.design_version == "B" ? 1 : 0,  // dummy vari
able
            interaction.time_on_page / 60,              // хвилини
            interaction.scroll_depth / 100,             // частка
            interaction.clicks,
            // взаємодії
            (interaction.design_version == "B" ? 1 : 0) * interaction
.time_on_page / 60
        ]
```

```
        X.append(features)
        y.append(interaction.converted)

    return X, y

// Аналіз результатів
model = logisticRegression(X, y)

// Коефіцієнти:
// w[0] = 0.15  - базовий ефект дизайну B
// w[1] = 0.08  - вплив часу на сторінці
// w[2] = 0.12  - вплив глибини прокрутки
// w[3] = 0.05  - вплив кліків
// w[4] = 0.03  - додатковий ефект часу для дизайну B

// Статистична значущість
p_value_design_B = computePValue(model, feature_index=0)
// p_value = 0.02 < 0.05 - статистично значущий

// Висновок: Дизайн B збільшує ймовірність конверсії на 15%
```

Переваги та недоліки

Переваги

- Інтерпретованість: Коефіцієнти показують вплив кожної ознаки на log-odds, що дозволяє зрозуміти важливість факторів.

- Ймовірнісні прогнози: Надає не тільки клас, але й ймовірність, що корисно для прийняття рішень.

- Ефективність: Швидке навчання та прогнозування, підходить для великих датасетів.

- Не потребує налаштування: Мало гіперпараметрів порівняно з іншими методами.

- Працює з малими даними: Може давати хороші результати навіть на невеликих вибірках.

- Стійкість до шуму: Менш схильна до перенавчання порівняно з деревами рішень.

Недоліки

- Лінійність: Передбачає лінійну границю розділення класів, не може моделювати складні нелінійні залежності.

- Чутливість до викидів: Екстремальні значення можуть сильно вплинути на результат.

- Незбалансовані класи: Погано працює коли один клас значно переважає інший без додаткових технік.

- Мультиколінеарність: Проблеми при сильній кореляції між ознаками.

- Feature engineering: Потребує ручного створення ознак для моделювання нелінійностей.

- Припущення про незалежність: Передбачає умовну незалежність ознак при заданому класі.

Контрольні запитання

1. Чим логістична регресія відрізняється від лінійної регресії?

2. Що таке логістична функція та які її основні властивості?

3. Яка функція втрат використовується в логістичній регресії та чому?

4. Як інтерпретувати коефіцієнти логістичної регресії?

5. Що таке odds та log-odds в контексті логістичної регресії?

6. Як працює багатокласова логістична регресія?

7. Які методи регуляризації застосовуються в логістичній регресії?

8. Що таке ROC-крива та AUC?

9. Як вибрати оптимальний поріг для класифікації?

10. Які припущення робить логістична регресія про дані?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритмів логістичної регресії

2. Мета

Вивчити теоретичні основи логістичної регресії, реалізувати різні методи оптимізації, навчитися оцінювати якість класифікації, порівняти з іншими методами машинного навчання.

3. Завдання

Реалізувати бінарну та багатокласову логістичну регресію, різні методи оптимізації (градієнтний спуск, стохастичний градієнт, метод Ньютона), додати регуляризацію, створити систему оцінки якості з ROC-аналізом.

4. Опис

Лабораторна робота охоплює повний цикл розробки класифікатора: від підготовки даних до аналізу результатів. Студенти реалізують основні алгоритми, методи візуалізації границь рішення та оцінки якості.

5. Порядок виконання

- Реалізувати сигмоїдну функцію та її похідну
- Створити функцію обчислення бінарної крос-ентропії
- Реалізувати градієнтний спуск для логістичної регресії
- Додати стохастичний та міні-batch градієнтний спуск
- Реалізувати L1 та L2 регуляризацію
- Створити багатокласову версію з softmax
- Реалізувати метрики оцінки (accuracy, precision, recall, F1)
- Побудувати ROC-криву та обчислити AUC
- Візуалізувати границі рішення для 2D даних
- Порівняти з іншими класифікаторами

6. Результат

Повнофункціональна бібліотека логістичної регресії з підтримкою різних режимів, система візуалізації результатів класифікації, детальний аналіз продуктивності на різних датасетах, рекомендації щодо застосування.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас LogisticRegression з методами fit, predict, predict_proba
- Реалізація різних оптимізаторів
- Функції для багатокласової класифікації
- Модуль метрик та візуалізації
- Cross-validation для вибору гіперпараметрів

2. Продемонструвати роботу додатку:

- Класифікація на синтетичних 2D даних
- Візуалізація процесу навчання (зміна втрат)
- Порівняння різних методів оптимізації
- ROC-аналіз та вибір оптимального порогу
- Робота з реальними даними

3. Бути готовим відповісти на питання:

- Коли логістична регресія працює добре/погано?
- Як вибрати learning rate та параметр регуляризації?
- Що робити при незбалансованих класах?
- Як інтерпретувати коефіцієнти моделі?

Тести

1. Логістична функція відображає:

- 1) (-∞, +∞) → (-1, 1)
- 2) (-∞, +∞) → (0, 1)
- 3) (0, 1) → (-∞, +∞)
- 4) (-1, 1) → (0, 1)

2. Функція втрат для логістичної регресії:

- 1) Середньоквадратична похибка
- 2) Абсолютна похибка
- 3) Бінарна крос-ентропія
- 4) Hinge loss

3. Softmax функція використовується для:

- 1) Бінарної класифікації
- 2) Регресії
- 3) Багатокласової класифікації
- 4) Кластеризації

4. Що означає AUC = 0.5?

- 1) Ідеальний класифікатор
- 2) Випадкове вгадування
- 3) Найгірший класифікатор
- 4) Помилка в розрахунках

5. L1 регуляризація в логістичній регресії:

- 1) Збільшує всі коефіцієнти
- 2) Може обнулити деякі коефіцієнти
- 3) Не впливає на коефіцієнти

- 4) Робить всі коефіцієнти однаковими

Відповіді:

1-2,

2-3,

3-3,

4-2,

5-2

Тести підвищеної складності

1. Чому не використовують MSE для логістичної регресії?

- 1) Занадто складно обчислити

- 2) Функція втрат стає неопуклою

- 3) Не працює для класифікації

- 4) Потребує більше пам’яті

2. Що таке логарифмічні шанси (log-odds)?

- 1) log(p)

- 2) log(1-p)

- 3) log(p/(1-p))

- 4) log(p*(1-p))

3. Коли варто використовувати метод Ньютона замість градієнтного спуску?

- 1) Завжди

- 2) Для великих датасетів

- 3) Коли важлива швидка збіжність і датасет невеликий

- 4) Ніколи

4. Що означає висока precision але низький recall?

- 1) Модель часто помиляється

- 2) Модель консервативна, рідко передбачає позитивний клас

- 3) Модель передбачає занадто багато позитивних

- 4) Модель ідеальна

5. Як впливає незбалансованість класів на логістичну регресію?

- 1) Не впливає

- 2) Зміщує границю рішення до меншого класу

- 3) Покращує якість

- 4) Робить навчання неможливим

6. Що таке focal loss?

- 1) Звичайна крос-ентропія

- 2) Модифікована втрата для незбалансованих даних

- 3) L2 регуляризація

- 4) Метрика якості

7. Чому градієнт логістичної регресії має просту форму?

- 1) Це збіг

- 2) Завдяки властивостям похідної сигмоїди

- 3) Через лінійність моделі

- 4) Це неправда

8. Коли мультиноміальна регресія еквівалентна набору бінарних one-vs-rest?

- 1) Завжди

- 2) Ніколи

- 3) Коли класи лінійно розділені

- 4) Коли використовується softmax

9. Що таке calibration в контексті логістичної регресії?

- 1) Налаштування гіперпараметрів

- 2) Відповідність передбачених ймовірностей реальним частотам

- 3) Нормалізація ознак

- 4) Вибір порогу

10. Як ordinal regression відрізняється від звичайної багатокласової?

- 1) Не відрізняється

- 2) Враховує порядок класів

- 3) Працює швидше

- 4) Використовує інші ознаки

Відповіді: 1. b) MSE робить оптимізаційну задачу неопуклою для логістичної функції 2. c) Логарифм відношення ймовірностей p/(1-p) 3. c) Метод Ньютона швидший але вимагає обчислення Гессіана 4. b) Модель рідко каже “так”, але коли каже - зазвичай права 5. b) Більший клас домінує в функції втрат 6. b) Додає ваговий коефіцієнт для складних прикладів 7. b) σ’(z) = σ(z)(1-σ(z)) спрощує вирази 8. c) За певних умов підходи еквівалентні 9. b) Важлива властивість для прийняття рішень на основі ймовірностей 10. b) Використовує впорядкованість категорій (низький/середній/високий)

Завдання для самостійної роботи

1. Реалізуйте online логістичну регресію для потокових даних.
2. Створіть ансамбль логістичних регресій з агрегуванням.
3. Розробіть логістичну регресію для нелінійних границь.
4. Реалізуйте multi-label класифікацію.
5. Створіть адаптивний вибір темп навчання (AdaGrad, Adam).
6. Розробіть cost-sensitive версію для різної ціни помилок.
7. Реалізуйте довірчі інтервали для прогнозів.
8. Створіть поступове навчання з початковою ініціалізацією.
9. Розробіть розріджену логістичну регресію для високорозмірних даних.
10. Дослідіть Bayesian логістичну регресію.

Теми лабораторних робіт

1. Базова реалізація бінарної логістичної регресії.
2. Порівняння методів оптимізації для логістичної регресії.
3. Багатокласова класифікація: мультиноміальну класифікацію (softmax) і класифікацію «один проти всіх».
4. Робота з незбалансованими даними в класифікації.
5. Feature engineering для покращення логістичної регресії.
6. Інтерпретація моделей: важливість ознак та SHAP values.
7. Калібрування ймовірностей для прийняття рішень.
8. Логістична регресія для обробки тексту.
9. Порівняння логістичної регресії з нейронними мережами.
10. Застосування логістичної регресії в A/B тестуванні.

Кластеризація: K-Means

Теоретичні відомості

K-Means є одним з найпопулярніших та найпростіших алгоритмів кластеризації, який належить до класу методів навчання без учителя. Алгоритм був незалежно розроблений кількома дослідниками у 1950-1960-х роках, зокрема Стюартом Ллойдом у Bell Labs (1957) та Джеймсом МакКвіном (1967), який дав методу назву “K-Means”. Основна ідея алгоритму полягає в розбитті набору даних на K груп (кластерів) таким чином, щоб точки всередині кожного кластера були максимально схожі між собою та відрізнялися від точок інших кластерів.

Нехай маємо набір даних $X = \{x_1, x_2, \ldots, x_n\}$, де кожна точка $x_i \in \mathbb{R}^d$ є d-вимірним вектором ознак. Мета алгоритму K-Means - знайти K центроїдів $C = \{c_1, c_2, \ldots, c_K\}$ та призначення кожної точки до одного з K кластерів так, щоб мінімізувати внутрішньокластерну суму квадратів відстаней (within-cluster sum of squares, WCSS):

$$J = \sum_{k=1}^{K} \sum_{x_i \in S_k} ||x_i - c_k||^2$$

де: - S_k - множина точок, що належать k-му кластеру - c_k - центроїд k-го кластера - $||x_i - c_k||^2$ - квадрат евклідової відстані між точкою x_i та центроїдом c_k

Ця цільова функція також називається інерцією кластерів і показує, наскільки щільно згруповані точки навколо своїх центроїдів.

Центроїд (Centroid) - це точка в просторі ознак, яка представляє центр кластера. Математично центроїд обчислюється як середнє арифметичне всіх точок, що належать кластеру:

$$c_k = \frac{1}{|S_k|} \sum_{x_i \in S_k} x_i$$

де $|S_k|$ - кількість точок у k-му кластері.

Відстань - міра схожості між точками. У класичному K-Means використовується евклідова відстань:

$$d(x_i, x_j) = \sqrt{\sum_{l=1}^{d} (x_{il} - x_{jl})^2}$$

де x_{il} - l-та координата точки x_i.

Призначення до кластера - кожна точка призначається до найближчого центроїда:

$$s_i = \arg\min_{k \in \{1,\dots,K\}} ||x_i - c_k||^2$$

де s_i - номер кластера для точки x_i.

Алгоритм Lloyd's (стандартний K-Means)

Класичний алгоритм K-Means, також відомий як алгоритм Ллойда, працює ітеративно, чергуючи два кроки до досягнення збіжності:

1. Крок призначення (Assignment step): Кожна точка призначається до найближчого центроїда

2. Крок оновлення (Update step): Центроїди перераховуються як середні значення точок у відповідних кластерах

Детальний опис алгоритму K-Means:

```
struct KMeansData {
    X: matrix[n][d]     // матриця даних: n точок, d ознак
    n: integer          // кількість точок
    d: integer          // розмірність простору
}

struct KMeansModel {
    centroids: matrix[K][d]  // K центроїдів
    labels: vector[n]        // призначення кластерів для кожної точк
и
    inertia: float          // внутрішньокластерна сума квадратів
    n_iterations: integer    // кількість ітерацій до збіжності
}

function kMeans(data: KMeansData, K: integer, max_iterations: integer
,
                tolerance: float): KMeansModel
    // Крок 1: Ініціалізація центроїдів
    centroids = initializeCentroids(data, K)
    labels = vector[data.n]
    prev_inertia = infinity
```

```
    for iteration = 1 to max_iterations do
        // Крок 2: Призначення точок до кластерів
        new_labels = vector[data.n]
        total_distance = 0.0

        for i = 1 to data.n do
            min_distance = infinity
            closest_centroid = -1

            // Знаходимо найближчий центроїд
            for k = 1 to K do
                distance = euclideanDistance(data.X[i], centroids[k])
                if distance < min_distance then
                    min_distance = distance
                    closest_centroid = k

            new_labels[i] = closest_centroid
            total_distance += min_distance * min_distance

        // Крок 3: Оновлення центроїдів
        new_centroids = matrix[K][data.d]
        cluster_sizes = vector[K]

        // Обнуляємо суми
        for k = 1 to K do
            for j = 1 to data.d do
                new_centroids[k][j] = 0
            cluster_sizes[k] = 0

        // Накопичуємо суми для кожного кластера
        for i = 1 to data.n do
            cluster = new_labels[i]
            cluster_sizes[cluster] += 1
            for j = 1 to data.d do
                new_centroids[cluster][j] += data.X[i][j]

        // Обчислюємо середні значення
        for k = 1 to K do
            if cluster_sizes[k] > 0 then
                for j = 1 to data.d do
                    new_centroids[k][j] = new_centroids[k][j] / clust
er_sizes[k]
            else
                // Обробка порожнього кластера
                new_centroids[k] = reinitializeEmptyCluster(data, K,
```

```
new_labels)

        // Крок 4: Перевірка збіжності
        current_inertia = total_distance

        if abs(current_inertia - prev_inertia) < tolerance then
            // Алгоритм збігся
            return KMeansModel(new_centroids, new_labels, current_ine
rtia, iteration)

        centroids = new_centroids
        labels = new_labels
        prev_inertia = current_inertia

    // Досягнуто максимальну кількість ітерацій
    return KMeansModel(centroids, labels, prev_inertia, max_iteration
s)

function euclideanDistance(point1: vector[d], point2: vector[d]): flo
at
    distance = 0.0
    for j = 1 to d do
        diff = point1[j] - point2[j]
        distance += diff * diff
    return sqrt(distance)
```

Методи ініціалізації центроїдів

Вибір початкових центроїдів критично впливає на якість кластеризації та швидкість збіжності. Розглянемо основні методи:

Випадкова ініціалізація

```
function randomInitialization(data: KMeansData, K: integer): matrix[K
][d]
    centroids = matrix[K][data.d]

    // Вибираємо K випадкових точок як початкові центроїди
    selected_indices = randomSample(range(1, data.n), K)

    for k = 1 to K do
        centroids[k] = data.X[selected_indices[k]]

    return centroids
```

K-Means++ ініціалізація

K-Means++ - це розумніший метод ініціалізації, розроблений Девідом Артуром та Сергієм Васильвицьким у 2007 році, який вибирає початкові центроїди з ймовірністю, пропорційною квадрату відстані до найближчого вже вибраного центроїда:

```
function kMeansPlusPlusInitialization(data: KMeansData, K: integer): matrix[K][d]
    centroids = matrix[K][data.d]

    // Вибираємо перший центроїд випадково
    first_index = randomInteger(1, data.n)
    centroids[1] = data.X[first_index]

    // Вибираємо решту K-1 центроїдів
    for k = 2 to K do
        // Обчислюємо відстані до найближчого центроїда для кожної точки
        distances = vector[data.n]

        for i = 1 to data.n do
            min_distance = infinity

            for j = 1 to k-1 do
                distance = euclideanDistance(data.X[i], centroids[j])
                if distance < min_distance then
                    min_distance = distance

            distances[i] = min_distance * min_distance

        // Вибираємо наступний центроїд з ймовірністю пропорційною D(x)²
        probabilities = normalize(distances)
        next_index = weightedRandomChoice(range(1, data.n), probabilities)
        centroids[k] = data.X[next_index]

    return centroids

function normalize(values: vector): vector
    sum_values = sum(values)
    normalized = vector[length(values)]

    for i = 1 to length(values) do
        normalized[i] = values[i] / sum_values
```

```
    return normalized
```

Вибір оптимального K

Визначення оптимальної кількості кластерів є важливою проблемою. Розглянемо основні методи:

1. Метод ліктя (Elbow Method)

```
function elbowMethod(data: KMeansData, max_K: integer): integer
    inertias = vector[max_K]

    for k = 1 to max_K do
        model = kMeans(data, k, max_iterations=300, tolerance=1e-4)
        inertias[k] = model.inertia

    // Знаходимо "лікоть" - точку максимальної кривизни
    optimal_k = findElbowPoint(inertias)

    return optimal_k

function findElbowPoint(inertias: vector): integer
    n = length(inertias)

    // Обчислюємо другу похідну
    second_derivatives = vector[n-2]

    for i = 2 to n-1 do
        second_derivatives[i-1] = inertias[i+1] - 2*inertias[i] + ine
rtias[i-1]

    // Знаходимо максимум другої похідної
    max_index = 1
    max_value = second_derivatives[1]

    for i = 2 to n-2 do
        if second_derivatives[i] > max_value then
            max_value = second_derivatives[i]
            max_index = i

    return max_index + 1
```

2. Силует (Silhouette Score)

```
function silhouetteScore(data: KMeansData, labels: vector[n],
                         centroids: matrix[K][d]): float
    n = data.n
    silhouette_values = vector[n]
```

```
    for i = 1 to n do
        // a(i) - середня відстань до точок свого кластера
        cluster_i = labels[i]
        same_cluster_distances = []

        for j = 1 to n do
            if i != j and labels[j] == cluster_i then
                same_cluster_distances.append(euclideanDistance(data.
X[i], data.X[j]))

        a_i = mean(same_cluster_distances) if length(same_cluster_dis
tances) > 0 else 0

        // b(i) - мінімальна середня відстань до точок інших кластері
в
        b_i = infinity

        for k = 1 to K do
            if k != cluster_i then
                other_cluster_distances = []

                for j = 1 to n do
                    if labels[j] == k then
                        other_cluster_distances.append(
                            euclideanDistance(data.X[i], data.X[j]))

                if length(other_cluster_distances) > 0 then
                    avg_distance = mean(other_cluster_distances)
                    if avg_distance < b_i then
                        b_i = avg_distance

        // Силует для точки i
        if max(a_i, b_i) > 0 then
            silhouette_values[i] = (b_i - a_i) / max(a_i, b_i)
        else
            silhouette_values[i] = 0

    return mean(silhouette_values)
```

Варіації алгоритму K-Means

1. Mini-Batch K-Means

Для великих датасетів використовується Mini-Batch версія, яка оновлює центроїди на основі підвибірок даних:

```
function miniBatchKMeans(data: KMeansData, K: integer, batch_size: in
teger,
                         max_iterations: integer): KMeansModel
    centroids = kMeansPlusPlusInitialization(data, K)
    cluster_counts = vector[K]  // лічильники точок у кластерах

    for iteration = 1 to max_iterations do
        // Вибираємо випадковий batch
        batch_indices = randomSample(range(1, data.n), batch_size)

        // Призначення для batch
        for idx in batch_indices do
            point = data.X[idx]

            // Знаходимо найближчий центроїд
            closest = 1
            min_distance = euclideanDistance(point, centroids[1])

            for k = 2 to K do
                distance = euclideanDistance(point, centroids[k])
                if distance < min_distance then
                    min_distance = distance
                    closest = k

            // Оновлюємо центроїд інкрементально
            cluster_counts[closest] += 1
            learning_rate = 1.0 / cluster_counts[closest]

            for j = 1 to data.d do
                centroids[closest][j] = (1 - learning_rate) * centroi
ds[closest][j] +
                                        learning_rate * point[j]

    // Фінальне призначення всіх точок
    labels = assignAllPoints(data, centroids)
    inertia = computeInertia(data, centroids, labels)

    return KMeansModel(centroids, labels, inertia, max_iterations)
```

2. K-Medoids (PAM - Partitioning Around Medoids)

K-Medoids використовує реальні точки датасету як центри кластерів:

```
function kMedoids(data: KMeansData, K: integer, max_iterations: integ
er): KMedoidsModel
    n = data.n
```

```
    // Ініціалізація: вибираємо K випадкових точок як медоїди
    medoid_indices = randomSample(range(1, n), K)

    for iteration = 1 to max_iterations do
        // Крок призначення
        labels = vector[n]
        total_cost = 0.0

        for i = 1 to n do
            min_distance = infinity
            closest_medoid = -1

            for k = 1 to K do
                distance = euclideanDistance(data.X[i], data.X[medoid
_indices[k]])
                if distance < min_distance then
                    min_distance = distance
                    closest_medoid = k

            labels[i] = closest_medoid
            total_cost += min_distance

        // Крок оновлення медоїдів
        improved = false

        for k = 1 to K do
            current_medoid = medoid_indices[k]
            best_medoid = current_medoid
            best_cost = computeClusterCost(data, labels, k, current_m
edoid)

            // Перевіряємо всі точки кластера як потенційні медоїди
            for i = 1 to n do
                if labels[i] == k and i != current_medoid then
                    new_cost = computeClusterCost(data, labels, k, i)

                    if new_cost < best_cost then
                        best_cost = new_cost
                        best_medoid = i
                        improved = true

            medoid_indices[k] = best_medoid

        if not improved then
            break
```

```
    return KMedoidsModel(medoid_indices, labels, total_cost)
```

Обробка категоріальних даних

Для роботи з категоріальними ознаками використовується алгоритм K-Modes:

```
function kModes(data: CategoricalData, K: integer, max_iterations: in
teger): KModesModel
    n = data.n
    d = data.d

    // Ініціалізація модами (найчастішими значеннями)
    modes = initializeModes(data, K)

    for iteration = 1 to max_iterations do
        // Призначення на основі відстані Хеммінга
        labels = vector[n]

        for i = 1 to n do
            min_dissimilarity = infinity
            closest_mode = -1

            for k = 1 to K do
                dissimilarity = hammingDistance(data.X[i], modes[k])
                if dissimilarity < min_dissimilarity then
                    min_dissimilarity = dissimilarity
                    closest_mode = k

            labels[i] = closest_mode

        // Оновлення мод
        new_modes = matrix[K][d]

        for k = 1 to K do
            cluster_points = getClusterPoints(data, labels, k)

            if length(cluster_points) > 0 then
                // Знаходимо моду для кожної ознаки
                for j = 1 to d do
                    new_modes[k][j] = findMode(cluster_points, j)
            else
                // Реініціалізація порожнього кластера
                new_modes[k] = randomDataPoint(data)

        // Перевірка збіжності
```

```
        if matricesEqual(modes, new_modes) then
            break

        modes = new_modes

    return KModesModel(modes, labels)

function hammingDistance(point1: vector, point2: vector): integer
    distance = 0
    for j = 1 to length(point1) do
        if point1[j] != point2[j] then
            distance += 1
    return distance
```

Область застосування

K-Means широко використовується в різних галузях завдяки своїй простоті та ефективності:

- Сегментація клієнтів: Групування клієнтів за поведінкою, демографією, вподобаннями для таргетованого маркетингу.

- Обробка зображень: Сегментація зображень, стиснення кольорів, виявлення об’єктів.

- Аналіз документів: Кластеризація текстів за темами, організація великих колекцій документів.

- Біоінформатика: Групування генів за експресією, класифікація білків.

- Аномалії: Виявлення викидів як точок, далеких від всіх центроїдів.

- Попередня обробка: Зменшення розмірності даних, дискретизація неперервних змінних.

Приклади застосування

Приклад 1: Базова кластеризація 2D точок

Розглянемо простий приклад кластеризації точок на площині:

```
Дані (6 точок):
X = [[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11]]

K = 2, випадкова ініціалізація

Ініціалізація:
c1 = [1, 2] (перша точка)
c2 = [8, 8] (четверта точка)
```

```
Ітерація 1:
Крок призначення:
- d(X1, c1) = 0, d(X1, c2) = 9.22 → кластер 1
- d(X2, c1) = 0.36, d(X2, c2) = 8.96 → кластер 1
- d(X3, c1) = 7.21, d(X3, c2) = 3 → кластер 2
- d(X4, c1) = 9.22, d(X4, c2) = 0 → кластер 2
- d(X5, c1) = 1.4, d(X5, c2) = 10.77 → кластер 1
- d(X6, c1) = 11.66, d(X6, c2) = 3.16 → кластер 2

Кластери: S1 = {X1, X2, X5}, S2 = {X3, X4, X6}

Крок оновлення:
c1 = [(1+1.5+1)/3, (2+1.8+0.6)/3] = [1.17, 1.47]
c2 = [(5+8+9)/3, (8+8+11)/3] = [7.33, 9]

Ітерація 2:
Крок призначення:
- Перераховуємо відстані з новими центроїдами
- Призначення не змінюються

WCSS = 1.35 + 8.22 = 9.57

Алгоритм збігся за 2 ітерації

Фінальні кластери:
Кластер 1: {[1, 2], [1.5, 1.8], [1, 0.6]}
Кластер 2: {[5, 8], [8, 8], [9, 11]}
```

Приклад 2: Сегментація клієнтів інтернет-магазину

```
struct CustomerData {
    avgOrderValue: float    // середня сума замовлення
    ordersPerMonth: float   // кількість замовлень на місяць
    daysSinceLastOrder: integer  // днів з останнього замовлення
}

// Дані про 8 клієнтів (нормалізовані)
customers = [
    {0.2, 0.3, 0.8},   // рідкісний, малі замовлення
    {0.9, 0.8, 0.1},   // частий, великі замовлення
    {0.3, 0.2, 0.9},   // рідкісний, давно не купував
    {0.8, 0.9, 0.2},   // VIP клієнт
    {0.1, 0.1, 0.7},   // майже неактивний
    {0.7, 0.7, 0.3},   // активний, середні замовлення
    {0.4, 0.5, 0.5},   // середній клієнт
    {0.9, 0.6, 0.2}    // великі але рідкі замовлення
```

```
]

// K-Means з K=3
model = kMeans(customers, K=3)

// Результати:
Кластер 1: "Сплячі клієнти" - низька активність, давно не купували
  Центроїд: [0.2, 0.2, 0.8]
  Клієнти: 1, 3, 5

Кластер 2: "VIP клієнти" - часті великі замовлення
  Центроїд: [0.8, 0.75, 0.2]
  Клієнти: 2, 4, 6, 8

Кластер 3: "Середні клієнти" - помірна активність
  Центроїд: [0.4, 0.5, 0.5]
  Клієнти: 7

// Маркетингові стратегії:
- Кластер 1: Реактиваційні email-кампанії, знижки
- Кластер 2: Програма лояльності, ексклюзивні пропозиції
- Кластер 3: Стимулювання до більших покупок
```

Приклад 3: Стиснення зображення

K-Means для зменшення кількості кольорів:

```
function imageCompression(image: array[height][width][3], K: integer)
: array
    // Перетворюємо зображення в масив пікселів
    pixels = []

    for i = 1 to height do
        for j = 1 to width do
            pixels.append([image[i][j][0], image[i][j][1], image[i][j
][2]])

    // Кластеризація кольорів
    kmeans_data = KMeansData(pixels, length(pixels), 3)
    model = kMeans(kmeans_data, K)

    // Заміна кольорів центроїдами
    compressed_image = array[height][width][3]
    pixel_index = 0

    for i = 1 to height do
        for j = 1 to width do
```

```
            cluster = model.labels[pixel_index]
            compressed_image[i][j] = model.centroids[cluster]
            pixel_index += 1

    return compressed_image

// Приклад: стиснення з 16.7M кольорів до 16
original_size = 1024 * 768 * 3 * 8  // біт
compressed_size = 1024 * 768 * 4 + 16 * 3 * 8  // 4 біти на піксель +
палітра

compression_ratio = original_size / compressed_size  // ≈ 48:1
```

Приклад 4: Аналіз тексту - групування документів

```
// Представлення документів через TF-IDF вектори
documents = [
    "Machine learning algorithms data",
    "Neural networks deep learning",
    "Database systems SQL queries",
    "Data structures algorithms complexity",
    "Relational database management",
    "Supervised learning classification"
]

// Створення TF-IDF матриці
vocabulary = ["machine", "learning", "algorithms", "data", "neural",
              "networks", "deep", "database", "systems", "sql",
              "queries", "structures", "complexity", "relational",
              "management", "supervised", "classification"]

// TF-IDF вектори (спрощено)
tfidf_matrix = [
    [0.5, 0.5, 0.5, 0.5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0.4, 0, 0, 0.6, 0.6, 0.6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0.5, 0.5, 0.5, 0.5, 0, 0, 0, 0, 0, 0],
    [0, 0, 0.4, 0.4, 0, 0, 0, 0, 0, 0, 0, 0.6, 0.6, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0.5, 0, 0, 0, 0, 0, 0.7, 0.5, 0, 0],
    [0, 0.4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.6, 0.6]
]

// K-Means з K=2
model = kMeans(tfidf_matrix, K=2)

// Результати:
Кластер 1: "Machine Learning" - документи 1, 2, 6
```

```
Кластер 2: "Databases" - документи 3, 5
Документ 4: може бути в будь-якому кластері (містить елементи обох те
м)
```

Приклад 5: Виявлення аномалій у мережевому трафіку

```
struct NetworkTraffic {
    packetsPerSecond: float
    bytesPerPacket: float
    portDiversity: float    // кількість унікальних портів / загальна
кількість
    protocolDistribution: float  // частка TCP vs UDP
}

function detectAnomalies(traffic_data: array of NetworkTraffic,
                         contamination: float): array of boolean
    // Кластеризація нормального трафіку
    K = 3  // припускаємо 3 типи нормальної поведінки
    model = kMeans(traffic_data, K)

    // Обчислення порогу для аномалій
    distances = []

    for i = 1 to length(traffic_data) do
        point = traffic_data[i]
        cluster = model.labels[i]
        centroid = model.centroids[cluster]

        distance = euclideanDistance(point, centroid)
        distances.append(distance)

    // Поріг на основі персентиля
    threshold = percentile(distances, 100 * (1 - contamination))

    // Позначення аномалій
    anomalies = []

    for i = 1 to length(distances) do
        anomalies[i] = distances[i] > threshold

    return anomalies

// Приклад виявлених аномалій:
// - DDoS атака: дуже високий packetsPerSecond, низький bytesPerPacke
t
```

```
// - Port scan: нормальний трафік але дуже високий portDiversity
// - Data exfiltration: високий bytesPerPacket, незвичайні порти
```

Переваги та недоліки

Переваги

- Простота: Легкий для розуміння та реалізації алгоритм.

- Швидкість: O(n·K·d·t), де t - кількість ітерацій, зазвичай невелика.

- Масштабованість: Добре працює з великими датасетами, особливо Mini-Batch версія.

- Гарантована збіжність: Завжди збігається до локального мінімуму.

- Інтерпретованість: Центроїди мають чітку інтерпретацію.

- Паралелізація: Легко розпаралелюється для прискорення.

Недоліки

- Вибір K: Потрібно заздалегідь знати кількість кластерів.

- Форма кластерів: Передбачає сферичні кластери однакового розміру.

- Чутливість до ініціалізації: Результат залежить від початкових центроїдів.

- Чутливість до викидів: Викиди сильно впливають на центроїди.

- Локальні мінімуми: Не гарантує знаходження глобального оптимуму.

- Масштаб ознак: Чутливий до різних масштабів ознак, потребує нормалізації.

Контрольні запитання

1. Що таке кластеризація та чим вона відрізняється від класифікації?

2. Яка математична задача оптимізації вирішується в алгоритмі K-Means?

3. Опишіть два основні кроки алгоритму K-Means.

4. Чому важлива ініціалізація центроїдів та як працює K-Means++?

5. Які існують методи вибору оптимального значення K?

6. Що таке інерція кластерів та як вона використовується?

7. Як обробляти категоріальні дані в задачах кластеризації?

8. Чим K-Medoids відрізняється від K-Means?

9. Які припущення робить K-Means про форму кластерів?

10. Як виявляти аномалії за допомогою K-Means?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму кластеризації K-Means

2. Мета

Вивчити принципи кластерного аналізу, реалізувати алгоритм K-Means з різними методами ініціалізації, навчитися оцінювати якість кластеризації, дослідити вплив параметрів на результат.

3. Завдання

Реалізувати базовий алгоритм K-Means, методи ініціалізації (випадкова, K-Means++), критерії вибору K (elbow, silhouette), варіації алгоритму (Mini-Batch, K-Medoids), візуалізацію результатів кластеризації.

4. Опис

Лабораторна робота охоплює повний цикл кластерного аналізу: від попередньої обробки даних до інтерпретації результатів. Студенти реалізують різні версії алгоритму та методи оцінки якості.

5. Порядок виконання

- Реалізувати обчислення евклідової відстані та інших метрик
- Створити базовий алгоритм K-Means з випадковою ініціалізацією
- Додати K-Means++ ініціалізацію
- Реалізувати метод ліктя для вибору K
- Обчислити силует для оцінки якості
- Створити Mini-Batch версію для великих даних
- Реалізувати візуалізацію кластерів для 2D/3D даних
- Додати обробку порожніх кластерів
- Провести експерименти з різними датасетами
- Порівняти з ієрархічною кластеризацією

6. Результат

Бібліотека кластерного аналізу з підтримкою різних алгоритмів, система автоматичного вибору оптимальних параметрів, візуалізація процесу кластеризації, аналіз стійкості результатів, практичні рекомендації.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас KMeans з методами fit, predict, fit_predict
- Реалізація різних методів ініціалізації
- Функції для вибору оптимального K
- Метрики оцінки якості кластеризації
- Візуалізація результатів

2. Продемонструвати роботу додатку:
- Кластеризація синтетичних даних різної форми
- Аналіз збіжності алгоритму
- Порівняння методів ініціалізації
- Вибір оптимального K різними методами
- Робота з реальними даними

3. Бути готовим відповісти на питання:
- Коли K-Means працює добре/погано?
- Як нормалізація впливає на результат?
- Що робити з порожніми кластерами?
- Як інтерпретувати результати кластеризації?

Тести

1. K-Means мінімізує:
- 1) Міжкластерну дисперсію
- 2) Внутрішньокластерну суму квадратів
- 3) Кількість кластерів
- 4) Відстань між центроїдами

2. Складність одної ітерації K-Means:
- 1) $O(n^2)$
- 2) $O(K^2)$
- 3) $O(n \cdot K \cdot d)$
- 4) $O(n \cdot \log n)$

3. K-Means++ покращує:
- 1) Швидкість збіжності
- 2) Якість ініціалізації

- 3) Форму кластерів
- 4) Кількість ітерацій

4. Силует приймає значення:
- 1) [0, 1]
- 2) [-1, 1]
- 3) [0, ∞)
- 4) (-∞, ∞)

5. K-Medoids використовує як центри:
- 1) Середні значення
- 2) Випадкові точки
- 3) Реальні точки даних
- 4) Нулі

Відповіді:

1-2,

2-3,

3-2,

4-2,

5-3

Тести підвищеної складності

1. Чому K-Means може не знайти глобальний оптимум?
- 1) Алгоритм працює неправильно
- 2) Цільова функція неопукла щодо призначень кластерів
- 3) Недостатньо ітерацій
- 4) Погана реалізація

2. Коли K-Means еквівалентний EM-алгоритму?
- 1) Ніколи
- 2) Для гаусових сумішей зі сферичними коваріаціями
- 3) Завжди
- 4) Для категоріальних даних

3. Що відбувається при K > n?

- 1) Алгоритм не працює
- 2) Кожна точка стає окремим кластером
- 3) Деякі кластери порожні
- 4) Збільшується точність

4. Яка відстань використовується в K-Modes?
- 1) Евклідова
- 2) Манхеттенська
- 3) Хеммінга
- 4) Косинусна

5. Чому нормалізація важлива для K-Means?
- 1) Прискорює обчислення
- 2) Ознаки з більшим масштабом домінують у відстані
- 3) Зменшує K
- 4) Не важлива

6. Що таке “curse of dimensionality” для K-Means?
- 1) Повільна робота
- 2) Всі точки стають рівновіддаленими
- 3) Потрібно більше K
- 4) Не впливає

7. Як K-Means пов’язаний з квантуванням векторів?
- 1) Не пов’язаний
- 2) K-Means є методом векторного квантування
- 3) Протилежні методи
- 4) K-Means використовує квантування

8. Коли використовувати косинусну відстань?
- 1) Для негативних даних
- 2) Коли важливий напрямок, а не величина
- 3) Ніколи
- 4) Для категоріальних даних

9. Що таке Gap статистика?

- 1) Відстань між кластерами
- 2) Метод вибору K через порівняння з випадковими даними
- 3) Помилка кластеризації
- 4) Швидкість збіжності

10. Як Fuzzy C-Means відрізняється від K-Means?
- 1) Використовує інші центроїди
- 2) Дозволяє часткове членство в кластерах
- 3) Працює швидше
- 4) Не потребує K

Відповіді:

1. 2) Проблема комбінаторної оптимізації з локальними мінімумами
2. 2) K-Means - окремий випадок EM для специфічних припущень
3. 3) Максимум n непорожніх кластерів
4. 3) Кількість відмінностей для категоріальних даних
5. 2) Без нормалізації домінують ознаки з великими значеннями
6. 2) У високих розмірностях відстані стають неінформативними
7. 2) Центроїди є кодовою книгою для квантування
8. 2) Для текстів, де важлива тематика, а не довжина
9. 2) Порівнює WCSS реальних та випадкових даних
10. 2) Кожна точка має ступені належності до всіх кластерів

Завдання для самостійної роботи

1. Реалізуйте K-Means з різними метриками відстані (Манхеттен, Чебишев).
2. Створіть адаптивний K-Means, який автоматично визначає K.
3. Розробіть kernel K-Means для нелінійно розділених кластерів.
4. Реалізуйте потоковий K-Means для онлайн кластеризації.
5. Створіть ієрархічний K-Means для великих K.
6. Розробіть constrained K-Means з обмеженнями на розміри кластерів.
7. Реалізуйте bisecting K-Means.
8. Створіть паралельну версію для GPU.
9. Розробіть робасну версію, стійку до викидів.

10. Дослідіть спектральну кластеризацію як альтернативу.

Теми лабораторних робіт

1. Базова реалізація K-Means та візуалізація.
2. Порівняння методів ініціалізації центроїдів.
3. Дослідження методів вибору оптимального K.
4. K-Means для сегментації зображень.
5. Кластеризація текстових документів.
6. Порівняння K-Means, K-Medoids та K-Modes.
7. Mini-Batch K-Means для великих даних.
8. Ансамблева кластеризація на основі K-Means.
9. Застосування для стиснення даних.
10. Гібридні методи: K-Means + DBSCAN.

Кластеризація: DBSCAN

Теоретичні відомості

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) є алгоритмом кластеризації на основі щільності, розробленим Мартіном Естером, Гансом-Петером Крігелем, Йоргом Сандером та Сяовеєм Сюй у 1996 році. На відміну від алгоритмів розбиття, таких як K-Means, DBSCAN не вимагає попереднього визначення кількості кластерів і може знаходити кластери довільної форми, а також ефективно виявляти викиди.

Фундаментальна ідея DBSCAN полягає в тому, що точки всередині кластера мають вищу щільність сусідів, ніж точки на межі кластерів або шумові точки. Алгоритм визначає кластери як області високої щільності, розділені областями низької щільності. Це дозволяє знаходити кластери складної форми, які не можуть бути виявлені методами, що припускають опуклі кластери.

Ключові концепції

ε-околиця (epsilon-neighborhood) точки p - це множина всіх точок, відстань до яких від p не перевищує ε:

$$N_\varepsilon(p) = \{q \in D | dist(p, q) \leq \varepsilon\}$$

де D - набір даних, dist(p, q) - функція відстані (зазвичай евклідова).

Основна точка (core point) - точка p є основною, якщо її ε-околиця містить принаймні MinPts точок (включаючи саму p):

$$|N_\varepsilon(p)| \geq MinPts$$

Гранична точка (border point) - точка, яка не є основною, але знаходиться в ε-околиці принаймні однієї основної точки.

Шумова точка (noise point) - точка, яка не є ні основною, ні граничною.

Пряма досяжність за щільністю (directly density-reachable) - точка q прямо досяжна за щільністю від точки p, якщо: - p є основною точкою - q належить ε-околиці p: $q \in N_\varepsilon(p)$

Досяжність за щільністю (density-reachable) - точка q досяжна за щільністю від точки p, якщо існує ланцюг точок $p_1, p_2, \ldots, p_n$, де $p_1 = p$ та $p_n = q$, такий що p_{i+1} прямо досяжна за щільністю від p_i для всіх i.

Зв'язність за щільністю (density-connected) - точки p та q зв'язані за щільністю, якщо існує точка o, від якої обидві точки p та q досяжні за щільністю.

Визначення кластера

У термінах DBSCAN, кластер C визначається як максимальна множина точок, зв'язаних за щільністю: - Для будь-яких двох точок p, q: якщо p ∈ C і q досяжна за щільністю від p, то q ∈ C - Для будь-яких двох точок p, q ∈ C: p та q зв'язані за щільністю

Розглянемо повну реалізацію алгоритму DBSCAN:

```
struct DBSCANData {
    X: matrix[n][d]     // матриця даних: n точок, d ознак
    n: integer          // кількість точок
    d: integer          // розмірність простору
}

struct DBSCANModel {
    labels: vector[n]        // мітки кластерів (-1 для шуму)
    core_points: set         // індекси основних точок
    n_clusters: integer      // кількість знайдених кластерів
    n_noise: integer         // кількість шумових точок
}

function DBSCAN(data: DBSCANData, eps: float, minPts: integer): DBSCA
NModel
    // Ініціалізація
    labels = vector[data.n]
```

```
    for i = 1 to data.n do
        labels[i] = UNDEFINED  // -2 означає невідвідану точку

    core_points = set()
    cluster_id = 0

    // Основний цикл алгоритму
    for i = 1 to data.n do
        if labels[i] != UNDEFINED then
            continue  // точка вже оброблена

        // Знаходимо всіх сусідів точки i
        neighbors = getNeighbors(data, i, eps)

        if length(neighbors) < minPts then
            // Позначаємо як шум (попередньо)
            labels[i] = NOISE  // -1
        else
            // i є основною точкою, починаємо новий кластер
            core_points.add(i)
            cluster_id += 1
            expandCluster(data, labels, i, neighbors, cluster_id, eps
, minPts, core_points)

    // Підрахунок статистики
    n_clusters = cluster_id
    n_noise = count(labels[i] == NOISE for i in 1..data.n)

    return DBSCANModel(labels, core_points, n_clusters, n_noise)

function getNeighbors(data: DBSCANData, point_idx: integer, eps: floa
t): list
    neighbors = []

    for j = 1 to data.n do
        if euclideanDistance(data.X[point_idx], data.X[j]) <= eps the
n
            neighbors.append(j)

    return neighbors

function expandCluster(data: DBSCANData, labels: vector, point_idx: i
nteger,
                       neighbors: list, cluster_id: integer, eps: floa
t,
```

```
                    minPts: integer, core_points: set)
    // Призначаємо поточну точку до кластера
    labels[point_idx] = cluster_id

    // Використовуємо чергу для обходу всіх досяжних точок
    seed_set = []
    for neighbor_idx in neighbors do
        if neighbor_idx != point_idx then
            seed_set.append(neighbor_idx)

    i = 0
    while i < length(seed_set) do
        current_point = seed_set[i]

        if labels[current_point] == NOISE then
            // Змінюємо шум на граничну точку
            labels[current_point] = cluster_id
        else if labels[current_point] == UNDEFINED then
            // Призначаємо невідвідану точку до кластера
            labels[current_point] = cluster_id

            // Знаходимо сусідів поточної точки
            current_neighbors = getNeighbors(data, current_point, eps
)

            if length(current_neighbors) >= minPts then
                // current_point є основною точкою
                core_points.add(current_point)

                // Додаємо її сусідів до seed_set
                for neighbor_idx in current_neighbors do
                    if neighbor_idx not in seed_set then
                        seed_set.append(neighbor_idx)

        i += 1

function euclideanDistance(point1: vector[d], point2: vector[d]): flo
at
    distance = 0.0
    for j = 1 to d do
        diff = point1[j] - point2[j]
        distance += diff * diff
    return sqrt(distance)
```

Оптимізація пошуку сусідів

Для великих датасетів пошук всіх сусідів для кожної точки має квадратичну складність. Використання просторових структур даних значно прискорює алгоритм:

```
struct KDTree {
    root: KDNode
    dimension: integer
}

struct KDNode {
    point: vector[d]
    point_idx: integer
    left: KDNode
    right: KDNode
    split_dim: integer
    split_value: float
}

function buildKDTree(data: DBSCANData): KDTree
    indices = range(1, data.n)
    root = buildKDTreeRecursive(data, indices, 0)
    return KDTree(root, data.d)

function buildKDTreeRecursive(data: DBSCANData, indices: list, depth:
integer): KDNode
    if length(indices) == 0 then
        return null

    if length(indices) == 1 then
        return KDNode(data.X[indices[1]], indices[1], null, null, -1,
0)

    // Вибираємо вимір для розділення
    split_dim = depth mod data.d

    // Сортуємо точки за обраним виміром
    sorted_indices = sortByDimension(data, indices, split_dim)
    median_idx = length(sorted_indices) / 2

    node = KDNode(
        point = data.X[sorted_indices[median_idx]],
        point_idx = sorted_indices[median_idx],
        split_dim = split_dim,
        split_value = data.X[sorted_indices[median_idx]][split_dim]
    )
```

```
    // Рекурсивно будуємо ліве та праве піддерева
    left_indices = sorted_indices[1:median_idx]
    right_indices = sorted_indices[median_idx+1:end]

    node.left = buildKDTreeRecursive(data, left_indices, depth + 1)
    node.right = buildKDTreeRecursive(data, right_indices, depth + 1)

    return node

function rangeQuery(tree: KDTree, query_point: vector[d], eps: float)
: list
    neighbors = []
    rangeQueryRecursive(tree.root, query_point, eps, neighbors, tree.
dimension)
    return neighbors

function rangeQueryRecursive(node: KDNode, query_point: vector[d], ep
s: float,
                             neighbors: list, dimension: integer)
    if node == null then
        return

    // Перевіряємо поточну точку
    distance = euclideanDistance(node.point, query_point)
    if distance <= eps then
        neighbors.append(node.point_idx)

    // Визначаємо, які піддерева потрібно перевірити
    if node.split_dim != -1 then
        diff = query_point[node.split_dim] - node.split_value

        if diff - eps <= 0 then
            rangeQueryRecursive(node.left, query_point, eps, neighbor
s, dimension)

        if diff + eps >= 0 then
            rangeQueryRecursive(node.right, query_point, eps, neighbo
rs, dimension)
```

Варіації та розширення DBSCAN

1. OPTICS (Ordering Points To Identify the Clustering Structure)

OPTICS є узагальненням DBSCAN, який створює впорядкування точок для виявлення кластерів при різних значеннях ε:

```
struct OPTICSPoint {
    index: integer
    reachability_distance: float
    core_distance: float
    processed: boolean
}

function OPTICS(data: DBSCANData, eps: float, minPts: integer): list
    points = []
    for i = 1 to data.n do
        points.append(OPTICSPoint(i, UNDEFINED, UNDEFINED, false))

    ordered_list = []

    for i = 1 to data.n do
        if not points[i].processed then
            expandClusterOrder(data, points, i, eps, minPts, ordered_
list)

    return ordered_list

function expandClusterOrder(data: DBSCANData, points: list, point_idx
: integer,
                            eps: float, minPts: integer, ordered_list:
list)
    neighbors = getNeighbors(data, point_idx, eps)
    points[point_idx].processed = true

    core_dist = coreDistance(data, point_idx, minPts, eps)
    points[point_idx].core_distance = core_dist

    ordered_list.append(points[point_idx])

    if core_dist != UNDEFINED then
        seeds = PriorityQueue()  // впорядкована за reachability dist
ance
        update(data, neighbors, point_idx, seeds, points, eps, minPts
)

        while not seeds.isEmpty() do
            current = seeds.extractMin()
            neighbors_current = getNeighbors(data, current.index, eps
)
            points[current.index].processed = true
```

```
            core_dist_current = coreDistance(data, current.index, min
Pts, eps)
            points[current.index].core_distance = core_dist_current

            ordered_list.append(points[current.index])

            if core_dist_current != UNDEFINED then
                update(data, neighbors_current, current.index, seeds,
points, eps, minPts)

function coreDistance(data: DBSCANData, point_idx: integer, minPts: i
nteger, eps: float): float
    neighbors = getNeighbors(data, point_idx, eps)

    if length(neighbors) < minPts then
        return UNDEFINED

    // Сортуємо відстані до сусідів
    distances = []
    for neighbor_idx in neighbors do
        if neighbor_idx != point_idx then
            distances.append(euclideanDistance(data.X[point_idx], dat
a.X[neighbor_idx]))

    sort(distances)
    return distances[minPts - 1]  // minPts-та найменша відстань
```

2. HDBSCAN (Hierarchical DBSCAN)

HDBSCAN будує ієрархію кластерів і автоматично вибирає оптимальні кластери:

```
struct HDBSCANModel {
    labels: vector[n]
    probabilities: vector[n]     // ймовірності належності до кластер
ів
    outlier_scores: vector[n]    // оцінки аномальності
    condensed_tree: CondensedTree
    n_clusters: integer
}

function HDBSCAN(data: DBSCANData, min_cluster_size: integer,
                 min_samples: integer): HDBSCANModel
    // Крок 1: Обчислення взаємної досяжності
    core_distances = computeCoreDistances(data, min_samples)

    // Крок 2: Побудова мінімального кістякового дерева взаємної дося
```

```
жності
    mst = buildMutualReachabilityMST(data, core_distances)

    // Крок 3: Побудова ієрархії кластерів
    hierarchy = buildClusterHierarchy(mst)

    // Крок 4: Конденсація дерева кластерів
    condensed_tree = condenseTree(hierarchy, min_cluster_size)

    // Крок 5: Вилучення стабільних кластерів
    clusters = extractClusters(condensed_tree)

    // Крок 6: Призначення точок та обчислення ймовірностей
    labels, probabilities = assignPoints(condensed_tree, clusters)

    // Крок 7: Обчислення оцінок аномальності
    outlier_scores = computeOutlierScores(condensed_tree, labels)

    return HDBSCANModel(labels, probabilities, outlier_scores,
                        condensed_tree, length(clusters))

function computeMutualReachabilityDistance(data: DBSCANData, i: integ
er, j: integer,
                                           core_distances: vector): fl
oat
    distance = euclideanDistance(data.X[i], data.X[j])
    return max(core_distances[i], core_distances[j], distance)
```

2. Адаптивний DBSCAN

Для датасетів з різною локальною щільністю можна використовувати адаптивні версії:

```
function adaptiveDBSCAN(data: DBSCANData, k: integer, minPts: integer
): DBSCANModel
    // Обчислюємо k-distance для кожної точки
    k_distances = vector[data.n]

    for i = 1 to data.n do
        distances = []
        for j = 1 to data.n do
            if i != j then
                distances.append(euclideanDistance(data.X[i], data.X[
j]))

        sort(distances)
        k_distances[i] = distances[k]
```

```
    // Використовуємо локальний eps для кожної точки
    labels = vector[data.n]
    for i = 1 to data.n do
        labels[i] = UNDEFINED

    cluster_id = 0

    for i = 1 to data.n do
        if labels[i] != UNDEFINED then
            continue

        // Використовуємо k-distance як локальний eps
        local_eps = k_distances[i]
        neighbors = getNeighbors(data, i, local_eps)

        if length(neighbors) >= minPts then
            cluster_id += 1
            adaptiveExpandCluster(data, labels, i, neighbors, cluster
_id,
                                  k_distances, minPts)
        else
            labels[i] = NOISE

    return DBSCANModel(labels, core_points, cluster_id, count_noise(l
abels))
```

Вибір параметрів

1. Метод k-distance графіка

```
function suggestEps(data: DBSCANData, minPts: integer): float
    // Обчислюємо k-distance для всіх точок
    k_distances = []

    for i = 1 to data.n do
        distances = []
        for j = 1 to data.n do
            if i != j then
                distances.append(euclideanDistance(data.X[i], data.X[
j]))

        sort(distances)
        if length(distances) >= minPts - 1 then
            k_distances.append(distances[minPts - 1])

    // Сортуємо k-distances у спадному порядку
```

```
    sort(k_distances, descending=true)

    // Знаходимо "коліно" в графіку
    knee_point = findKneePoint(k_distances)

    return k_distances[knee_point]

function findKneePoint(sorted_distances: vector): integer
    n = length(sorted_distances)

    // Нормалізуємо координати
    x_norm = []
    y_norm = []

    for i = 1 to n do
        x_norm[i] = (i - 1) / (n - 1)
        y_norm[i] = (sorted_distances[i] - sorted_distances[n]) /
                    (sorted_distances[1] - sorted_distances[n])

    // Знаходимо точку максимальної відстані від прямої
    max_distance = 0
    knee_index = 1

    for i = 2 to n-1 do
        // Відстань від точки до прямої між першою та останньою точка
ми
        distance = abs((y_norm[n] - y_norm[1]) * x_norm[i] -
                       (x_norm[n] - x_norm[1]) * y_norm[i] +
                       x_norm[n] * y_norm[1] - y_norm[n] * x_norm[1])
/
                   sqrt((y_norm[n] - y_norm[1])^2 + (x_norm[n] - x_nor
m[1])^2)

        if distance > max_distance then
            max_distance = distance
            knee_index = i

    return knee_index
```

Область застосування

DBSCAN широко використовується в задачах, де кластери мають нерегулярну форму або різну щільність:

- Географічний аналіз: Виявлення районів з високою щільністю подій, кластеризація GPS-координат, аналіз розподілу населення.

- Аномалії в даних: Виявлення викидів у фінансових транзакціях, мережевому трафіку, сенсорних даних.

- Обробка зображень: Сегментація об’єктів складної форми, видалення шуму, розпізнавання текстур.

- Астрономія: Виявлення скупчень зірок, галактичних кластерів, аналіз розподілу космічних об’єктів.

- Біоінформатика: Кластеризація генетичних послідовностей, аналіз мікромасивів, виявлення функціональних модулів.

- Соціальні мережі: Виявлення спільнот, аналіз поширення інформації, ідентифікація впливових користувачів.

Приклади застосування

Приклад 1: Базова кластеризація 2D точок

Розглянемо простий приклад з двома кластерами різної форми:

```
Дані (12 точок):
X = [[1,1], [1,2], [2,1], [2,2],      // Кластер 1 (щільний квадрат)
     [8,8], [8,9], [9,8], [9,9],      // Кластер 2 (щільний квадрат)
     [5,5],                            // Шум
     [1,8], [2,8], [2,9]]              // Кластер 3 (трикутник)

Параметри: eps = 1.5, minPts = 3

Крок 1: Визначення типів точок
Точка [1,1]: сусіди = {[1,1], [1,2], [2,1]}, |сусіди| = 3 ≥ 3 → основ
на
Точка [1,2]: сусіди = {[1,1], [1,2], [2,1], [2,2]}, |сусіди| = 4 ≥ 3
→ основна
...
Точка [5,5]: сусіди = {[5,5]}, |сусіди| = 1 < 3 → не основна
Точка [1,8]: сусіди = {[1,8], [2,8]}, |сусіди| = 2 < 3 → не основна

Крок 2: Формування кластерів
Починаємо з точки [1,1] (основна):
- Кластер 1: expandCluster([1,1])
  - Додаємо [1,1] → мітка = 1
  - Сусіди [1,1]: [1,2], [2,1]
  - [1,2] основна → розширюємо, додаємо її сусідів
  - [2,1] основна → розширюємо
  - Результат: {[1,1], [1,2], [2,1], [2,2]}

Точка [8,8] (основна, ще не відвідана):
```

```
- Кластер 2: expandCluster([8,8])
  - Аналогічно формуємо {[8,8], [8,9], [9,8], [9,9]}

Точка [2,8] (основна, ще не відвідана):
- Кластер 3: expandCluster([2,8])
  - Додаємо [2,8] → мітка = 3
  - Сусіди: [1,8], [2,9]
  - [1,8] не основна → гранична точка кластера 3
  - [2,9] не основна → гранична точка кластера 3
  - Результат: {[1,8], [2,8], [2,9]}

Точка [5,5]: жоден кластер не досягає → шум (мітка = -1)

Фінальний результат:
Кластер 1: {[1,1], [1,2], [2,1], [2,2]}
Кластер 2: {[8,8], [8,9], [9,8], [9,9]}
Кластер 3: {[1,8], [2,8], [2,9]}
Шум: {[5,5]}
```

Приклад 2: Виявлення аномалій у фінансових транзакціях

```
struct Transaction {
    amount: float           // сума транзакції
    time_since_last: float  // час з попередньої транзакції (хвилини)
    merchant_risk: float    // ризик торговця (0-1)
}

function detectFraudulentTransactions(transactions: array of Transact
ion): array
    // Нормалізація даних
    data = normalizeTransactionData(transactions)

    // Параметри DBSCAN для виявлення аномалій
    // eps вибрано на основі аналізу нормальної поведінки
    eps = 0.3
    minPts = 5  // мінімум 5 схожих транзакцій для "нормальної" повед
інки

    model = DBSCAN(data, eps, minPts)

    // Транзакції позначені як шум є потенційно шахрайськими
    fraudulent = []
    for i = 1 to length(transactions) do
        if model.labels[i] == NOISE then
            risk_score = calculateRiskScore(transactions[i], data, mo
del)
```

```
            fraudulent.append({
                transaction: transactions[i],
                risk_score: risk_score,
                reason: analyzeAnomalyReason(transactions[i], data, m
odel)
            })

    return fraudulent

function analyzeAnomalyReason(transaction: Transaction, data: DBSCAND
ata,
                              model: DBSCANModel): string
    // Знаходимо найближчий кластер
    min_distance = infinity
    closest_cluster = -1

    for cluster_id = 1 to model.n_clusters do
        cluster_center = computeClusterCenter(data, model, cluster_id
)
        distance = euclideanDistance(transaction, cluster_center)

        if distance < min_distance then
            min_distance = distance
            closest_cluster = cluster_id

    // Аналізуємо відхилення
    if transaction.amount > getClusterMax(data, model, closest_cluste
r, "amount") * 2 then
        return "Надзвичайно велика сума транзакції"
    else if transaction.time_since_last < 1 then
        return "Підозріло швидка послідовність транзакцій"
    else if transaction.merchant_risk > 0.8 then
        return "Транзакція з високоризиковим торговцем"
    else
        return "Незвичайна комбінація параметрів"

// Приклад результату:
Транзакція #47: amount=$5000, time=0.5хв, risk=0.9
  → Шум (не належить жодному кластеру)
  → Risk Score: 0.95
  → Причина: "Надзвичайно велика сума + високоризиковий торговець"
```

Приклад 3: Сегментація зображення

DBSCAN для виділення об’єктів на зображенні:

```
function imageSegmentation(image: array[height][width][3],
                           spatial_weight: float): SegmentationResult
    // Створюємо 5D простір ознак: [x, y, R, G, B]
    data_points = []

    for i = 1 to height do
        for j = 1 to width do
            // Просторові координати з вагою
            x_weighted = i * spatial_weight
            y_weighted = j * spatial_weight

            // Кольорові компоненти (нормалізовані)
            r = image[i][j][0] / 255.0
            g = image[i][j][1] / 255.0
            b = image[i][j][2] / 255.0

            data_points.append([x_weighted, y_weighted, r, g, b])

    // DBSCAN кластеризація
    eps = 0.1  // підібрано експериментально
    minPts = 50  // мінімальний розмір сегмента

    dbscan_data = DBSCANData(data_points, length(data_points), 5)
    model = DBSCAN(dbscan_data, eps, minPts)

    // Створення сегментованого зображення
    segmented = array[height][width]
    point_idx = 0

    for i = 1 to height do
        for j = 1 to width do
            segmented[i][j] = model.labels[point_idx]
            point_idx += 1

    // Постобробка: об'єднання дрібних сегментів
    segmented = mergeSmallSegments(segmented, min_size=100)

    return SegmentationResult(segmented, model.n_clusters)

// Приклад: сегментація зображення неба з хмарами
// Результат:
// - Кластер 1: блакитне небо (великий зв'язний регіон)
// - Кластер 2-5: різні хмари (окремі білі області)
// - Шум: окремі пікселі на межах об'єктів
```

Приклад 4: Аналіз GPS траєкторій

Виявлення популярних місць зупинок:

```
struct GPSPoint {
    latitude: float
    longitude: float
    timestamp: integer
    user_id: integer
}

function findPopularLocations(gps_data: array of GPSPoint): array of
Location
    // Конвертуємо GPS координати в метричну систему
    projected_points = []

    for point in gps_data do
        x, y = latLonToMeters(point.latitude, point.longitude)
        projected_points.append([x, y])

    // DBSCAN для знаходження кластерів
    eps = 50  // 50 метрів
    minPts = 20  // мінімум 20 точок для популярного місця

    dbscan_data = DBSCANData(projected_points, length(projected_point
s), 2)
    model = DBSCAN(dbscan_data, eps, minPts)

    // Аналіз кластерів
    locations = []

    for cluster_id = 1 to model.n_clusters do
        cluster_points = getClusterPoints(gps_data, model.labels, clu
ster_id)

        // Обчислюємо центр кластера
        center_lat = mean([p.latitude for p in cluster_points])
        center_lon = mean([p.longitude for p in cluster_points])

        // Аналізуємо часові патерни
        time_distribution = analyzeTimeDistribution(cluster_points)

        // Визначаємо тип локації
        location_type = inferLocationType(time_distribution, cluster_
points)

        locations.append({
```

```
            latitude: center_lat,
            longitude: center_lon,
            type: location_type,
            popularity: length(cluster_points),
            peak_hours: time_distribution.peak_hours,
            unique_visitors: countUniqueUsers(cluster_points)
        })

    return sortByPopularity(locations)

function inferLocationType(time_dist: TimeDistribution, points: array
): string
    morning_ratio = time_dist.morning_visits / time_dist.total_visits
    evening_ratio = time_dist.evening_visits / time_dist.total_visits
    weekend_ratio = time_dist.weekend_visits / time_dist.total_visits
    avg_duration = computeAverageDuration(points)

    if morning_ratio > 0.6 and avg_duration < 30 then
        return "Кав'ярня / сніданок"
    else if evening_ratio > 0.5 and avg_duration > 60 then
        return "Ресторан / розваги"
    else if weekend_ratio > 0.7 then
        return "Парк / відпочинок"
    else if avg_duration > 240 then
        return "Офіс / робота"
    else
        return "Інше"
```

Приклад 5: Аналіз соціальних мереж

Виявлення спільнот на основі взаємодій:

```
struct UserInteraction {
    user1_id: integer
    user2_id: integer
    interaction_count: integer
    avg_response_time: float
    common_interests: integer
}

function detectCommunities(interactions: array of UserInteraction,
                           users: array of User): array of Community
    // Створюємо граф взаємодій
    n_users = length(users)
    adjacency_matrix = matrix[n_users][n_users]

    for interaction in interactions do
```

```
        weight = computeInteractionWeight(interaction)
        adjacency_matrix[interaction.user1_id][interaction.user2_id]
= weight
        adjacency_matrix[interaction.user2_id][interaction.user1_id]
= weight

    // Обчислюємо embedding користувачів через спектральний аналіз
    embedding = spectralEmbedding(adjacency_matrix, dimensions=10)

    // DBSCAN на embedding
    eps = selectEpsUsingElbow(embedding)
    minPts = 5  // мінімум 5 користувачів для спільноти

    dbscan_data = DBSCANData(embedding, n_users, 10)
    model = DBSCAN(dbscan_data, eps, minPts)

    // Формуємо спільноти
    communities = []

    for cluster_id = 1 to model.n_clusters do
        community_users = getUsersInCluster(users, model.labels, clus
ter_id)

        // Аналізуємо характеристики спільноти
        common_topics = findCommonTopics(community_users)
        activity_pattern = analyzeActivityPattern(community_users, in
teractions)
        cohesion = computeCommunityCohesion(community_users, adjacenc
y_matrix)

        communities.append({
            id: cluster_id,
            users: community_users,
            size: length(community_users),
            topics: common_topics,
            activity_level: activity_pattern.avg_interactions,
            cohesion_score: cohesion,
            description: generateCommunityDescription(common_topics,
activity_pattern)
        })

    // Аналіз ізольованих користувачів
    isolated_users = getUsersWithLabel(users, model.labels, NOISE)

    return {
```

```
        communities: communities,
        isolated_users: isolated_users,
        modularity: computeModularity(adjacency_matrix, model.labels)
    }

// Приклад результату:
Спільнота 1: "Любителі фотографії"
  - 127 користувачів
  - Основні теми: фотографія, камери, пейзажі
  - Високий рівень взаємодії (8.3 повідомлень/день)
  - Cohesion: 0.82

Спільнота 2: "Геймери"
  - 89 користувачів
  - Основні теми: ігри, стрімінг, турніри
  - Пікова активність: 20:00-02:00
  - Cohesion: 0.75

Ізольовані користувачі: 34 (потенційні боти або неактивні акаунти)
```

Переваги та недоліки

Переваги

- Довільна форма кластерів: Може знаходити кластери будь-якої форми, не обмежуючись сферичними.

- Автоматичне визначення кількості: Не потребує попереднього задання кількості кластерів.

- Виявлення шуму: Природно виділяє викиди та шумові точки.

- Детермінованість: За фіксованих параметрів завжди дає однаковий результат.

- Інтерпретованість: Параметри мають чітку фізичну інтерпретацію.

- Стійкість до викидів: Шумові точки не впливають на формування кластерів.

Недоліки

- Вибір параметрів: Чутливий до вибору eps та minPts, особливо для даних з різною щільністю.

- Складність $O(n^2)$: Без оптимізацій має квадратичну складність за кількістю точок.

- Проблема з різною щільністю: Погано працює коли кластери мають суттєво різну щільність.

- Високорозмірні дані: Ефективність знижується через “прокляття розмірності”.

- Граничні точки: Можуть бути призначені довільно при рівній відстані до кластерів.

- Масштабування ознак: Чутливий до різних масштабів ознак, потребує нормалізації.

Контрольні запитання

1. Що таке щільнісна кластеризація та чим вона відрізняється від кластеризації на основі центроїдів?

2. Поясніть концепції основної, граничної та шумової точок в DBSCAN.

3. Що означає досяжність за щільністю та зв’язність за щільністю?

4. Як вибрати оптимальні значення параметрів eps та minPts?

5. Яка часова складність DBSCAN та як її можна покращити?

6. Чим OPTICS відрізняється від DBSCAN?

7. Які переваги має HDBSCAN над класичним DBSCAN?

8. Як DBSCAN справляється з кластерами різної щільності?

9. Чому DBSCAN ефективний для виявлення аномалій?

10. Як адаптувати DBSCAN для потокових даних?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму кластеризації DBSCAN

2. Мета

Вивчити принципи щільнісної кластеризації, реалізувати алгоритм DBSCAN з оптимізаціями, навчитися вибирати параметри алгоритму, порівняти з іншими методами кластеризації.

3. Завдання

Реалізувати базовий алгоритм DBSCAN, додати оптимізацію через просторові структури даних, створити методи для вибору параметрів, реалізувати варіації (OPTICS, HDBSCAN), візуалізувати результати для різних типів даних.

4. Опис

Лабораторна робота охоплює реалізацію щільнісних методів кластеризації, аналіз їх поведінки на різних типах даних, методи оптимізації та практичне застосування для реальних задач.

5. Порядок виконання

- Реалізувати базовий DBSCAN з наївним пошуком сусідів
- Додати підтримку різних метрик відстані
- Реалізувати KD-дерево для прискорення пошуку
- Створити метод k-distance для вибору eps
- Реалізувати візуалізацію процесу кластеризації
- Додати OPTICS для аналізу ієрархії
- Провести експерименти з різними формами кластерів
- Реалізувати паралельну версію
- Порівняти з K-Means на різних датасетах
- Застосувати для реальної задачі (на вибір)

6. Результат

Повнофункціональна бібліотека щільнісної кластеризації з візуалізацією, система автоматичного підбору параметрів, аналіз продуктивності різних оптимізацій, порівняння з іншими методами, практичні рекомендації щодо застосування.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас DBSCAN з методами fit, fit_predict
- Реалізація просторових індексів
- Методи вибору параметрів
- Клас OPTICS для ієрархічного аналізу
- Візуалізація результатів та reachability plot

2. Продемонструвати роботу додатку:

- Кластеризація даних різної форми (місяці, кільця)
- Аналіз впливу параметрів на результат
- Порівняння часу роботи з/без оптимізацій
- Robustness до шуму та викидів

- Застосування до реальних даних

3. Бути готовим відповісти на питання:

- Коли використовувати DBSCAN замість K-Means?

- Як параметри впливають на результат?

- Що робити з даними різної щільності?

- Як масштабувати для великих датасетів?

Тести

1. DBSCAN належить до класу алгоритмів:

- 1) Ієрархічної кластеризації

- 2) Кластеризації на основі центроїдів

- 3) Щільнісної кластеризації

- 4) Спектральної кластеризації

2. Мінімальна кількість точок для основної точки визначається параметром:

- 1) eps

- 2) minPts

- 3) k

- 4) sigma

3. Точка є граничною, якщо:

- 1) Має менше minPts сусідів

- 2) Знаходиться в eps-околиці основної точки

- 3) Не має сусідів

- 4) Знаходиться на краю датасету

4. Часова складність наївного DBSCAN:

- 1) O(n)

- 2) O(n log n)

- 3) $O(n^2)$

- 4) $O(n^3)$

5. DBSCAN може знайти кластери:

- 1) Тільки сферичної форми

- 2) Тільки опуклої форми
- 3) Довільної форми
- 4) Тільки еліптичної форми

Відповіді:

1-3,

2-2,

3-2,

4-3,

5-3

Тести підвищеної складності

1. Чому DBSCAN детермінований, а K-Means - ні?
- 1) DBSCAN швидший
- 2) DBSCAN не має випадкової ініціалізації
- 3) DBSCAN використовує інші метрики
- 4) DBSCAN працює послідовно

2. Що станеться якщо eps занадто великий?
- 1) Багато шуму
- 2) Всі точки в одному кластері
- 3) Алгоритм не збіжиться
- 4) Порожні кластери

3. OPTICS зберігає інформацію про:
- 1) Центри кластерів
- 2) Reachability distance
- 3) Кількість ітерацій
- 4) Випадкові числа

4. Для прискорення DBSCAN найкраще використовувати:
- 1) Хеш-таблиці
- 2) Просторові індекси (KD-tree, R-tree)
- 3) Сортування
- 4) Кешування

5. HDBSCAN вирішує проблему:

- 1) Швидкості
- 2) Кластерів різної щільності
- 3) Високої розмірності
- 4) Категоріальних даних

6. Граничні точки можуть належати:

- 1) Тільки одному кластеру
- 2) Кільком кластерам (залежить від порядку)
- 3) Жодному кластеру
- 4) Всім кластерам

7. Для текстових даних з DBSCAN краще використовувати:

- 1) Евклідову відстань
- 2) Косинусну відстань
- 3) Манхеттенську відстань
- 4) Хеммінгову відстань

8. MinPts зазвичай вибирають:

- 1) = 1
- 2) ⩾ d + 1, де d - розмірність
- 3) = n/2
- 4) Випадково

9. Shared Nearest Neighbor (SNN) модифікація DBSCAN:

- 1) Прискорює алгоритм
- 2) Використовує схожість на основі спільних сусідів
- 3) Зменшує пам’ять
- 4) Додає ієрархію

10. Для потокових даних найкраще підходить:

- 1) Класичний DBSCAN
- 2) DenStream (потокова версія)
- 3) OPTICS
- 4) HDBSCAN

Відповіді:

1. 2) Відсутність випадкових елементів робить результат передбачуваним
2. 2) Занадто великий радіус об’єднає всі точки
3. 2) Відстань досяжності для побудови reachability plot
4. 2) Дерева дозволяють ефективний range query
5. 2) Ієрархічний підхід адаптується до локальної щільності
6. 2) Порядок обробки впливає на призначення граничних точок
7. 2) Для високорозмірних розріджених векторів
8. 2) Мінімум для статистичної значущості в d-вимірному просторі
9. 2) Покращує роботу з даними різної щільності
10. 2) Спеціально розроблений для онлайн кластеризації

Завдання для самостійної роботи

1. Реалізуйте Grid-based DBSCAN для прискорення на великих даних.
2. Створіть адаптивний DBSCAN з локальними параметрами eps.
3. Розробіть візуалізацію 3D кластерів з інтерактивним вибором параметрів.
4. Реалізуйте ST-DBSCAN для просторово-часових даних.
5. Створіть гібрид DBSCAN + K-Means для ієрархічної кластеризації.
6. Розробіть інкрементальний DBSCAN для динамічних даних.
7. Реалізуйте Fuzzy DBSCAN з м’якими межами кластерів.
8. Створіть Multi-density DBSCAN для складних датасетів.
9. Розробіть GPU-версію для масивних даних.
10. Дослідіть застосування DBSCAN для deep learning embeddings.

Теми лабораторних робіт

1. Базова реалізація DBSCAN та візуалізація.
2. Оптимізація через просторові структури даних.
3. Порівняння DBSCAN, OPTICS та HDBSCAN.
4. Автоматичний вибір параметрів DBSCAN.
5. Застосування для аналізу геопросторових даних.
6. Виявлення аномалій у часових рядах.

7. Кластеризація зображень та сегментація.

8. Аналіз соціальних мереж через DBSCAN.

9. Потокова кластеризація з DenStream.

10. Комбінування DBSCAN з іншими методами ML.

Ієрархічна кластеризація

Теоретичні відомості

Ієрархічна кластеризація є методом аналізу даних, який будує ієрархію кластерів у вигляді дерева. На відміну від методів розбиття, таких як K-Means, ієрархічна кластеризація не вимагає попереднього визначення кількості кластерів і надає повну картину можливих угруповань на різних рівнях деталізації. Цей підхід був розроблений у середині XX століття і став фундаментальним інструментом в багатьох галузях, від біології до соціальних наук.

Ієрархічна кластеризація створює вкладену послідовність кластерів, яка може бути представлена у вигляді дендрограми - деревоподібної діаграми, що показує порядок і рівень об’єднання кластерів. Існує два основних підходи до побудови ієрархії:

Агломеративний підхід (знизу вгору) - починає з того, що кожна точка даних є окремим кластером, і послідовно об’єднує найближчі кластери, поки всі точки не опиняться в одному кластері.

Дивізивний підхід (згори вниз) - починає з одного кластера, що містить всі точки, і рекурсивно розділяє кластери на менші підкластери.

Для побудови ієрархії необхідно визначити відстань між окремими точками та між кластерами. Відстань між точками зазвичай обчислюється за допомогою стандартних метрик:

Евклідова відстань:

$$d(x_i, x_j) = \sqrt{\sum_{k=1}^{d} (x_{ik} - x_{jk})^2}$$

Манхеттенська відстань:

$$d(x_i, x_j) = \sum_{k=1}^{d} |x_{ik} - x_{jk}|$$

Косинусна відстань:

$$d(x_i, x_j) = 1 - \frac{x_i \cdot x_j}{||x_i|| \cdot ||x_j||}$$

Методи зв'язку (Linkage Methods)

Ключовим аспектом ієрархічної кластеризації є визначення відстані між кластерами. Різні методи зв'язку дають різні результати кластеризації:

Одиничний зв'язок (Single Linkage) - відстань між кластерами визначається як мінімальна відстань між будь-якими двома точками з різних кластерів:

$$d(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y)$$

Повний зв'язок (Complete Linkage) - відстань між кластерами визначається як максимальна відстань між точками:

$$d(C_i, C_j) = \max_{x \in C_i, y \in C_j} d(x, y)$$

Середній зв'язок (Average Linkage) - використовує середню відстань між всіма парами точок:

$$d(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{x \in C_i} \sum_{y \in C_j} d(x, y)$$

Центроїдний зв'язок (Centroid Linkage) - відстань між центроїдами кластерів:

$$d(C_i, C_j) = d(\mu_i, \mu_j)$$

де $\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$ - центроїд кластера C_i.

Метод Варда (Ward's Method) - мінімізує внутрішньокластерну дисперсію:

$$d(C_i, C_j) = \sqrt{\frac{|C_i| \cdot |C_j|}{|C_i| + |C_j|}} \cdot d(\mu_i, \mu_j)$$

Агломеративна ієрархічна кластеризація

Розглянемо алгоритм агломеративної кластеризації:

```
struct HierarchicalData {
    X: matrix[n][d]     // матриця даних: n точок, d ознак
    n: integer          // кількість точок
    d: integer          // розмірність простору
```

```
}

struct Cluster {
    id: integer              // унікальний ідентифікатор кластера
    points: list of integer  // індекси точок у кластері
    centroid: vector[d]      // центроїд кластера
    height: float           // висота об'єднання в дендрограмі
    left_child: Cluster     // лівий дочірній кластер
    right_child: Cluster    // правий дочірній кластер
}

struct HierarchicalModel {
    root: Cluster           // корінь дерева кластерів
    dendrogram: list        // список об'єднань для побудови дендрогр
ами
    distance_matrix: matrix // матриця відстаней
}

function agglomerativeClustering(data: HierarchicalData,
                                 linkage_method: string): Hierarchical
Model
    // Крок 1: Ініціалізація - кожна точка є окремим кластером
    clusters = []
    for i = 1 to data.n do
        cluster = Cluster(
            id = i,
            points = [i],
            centroid = data.X[i],
            height = 0,
            left_child = null,
            right_child = null
        )
        clusters.append(cluster)

    // Крок 2: Обчислення початкової матриці відстаней
    distance_matrix = computeInitialDistanceMatrix(data)

    // Крок 3: Список для збереження історії об'єднань
    dendrogram = []
    next_cluster_id = data.n + 1

    // Крок 4: Основний цикл агломерації
    while length(clusters) > 1 do
        // Знаходимо пару найближчих кластерів
        min_distance = infinity
```

```
        merge_i = -1
        merge_j = -1

        for i = 1 to length(clusters) do
            for j = i+1 to length(clusters) do
                dist = computeClusterDistance(clusters[i], clusters[j
],
                                              distance_matrix, linkage_
method)
                if dist < min_distance then
                    min_distance = dist
                    merge_i = i
                    merge_j = j

        // Створюємо новий кластер об'єднанням двох найближчих
        new_cluster = Cluster(
            id = next_cluster_id,
            points = clusters[merge_i].points + clusters[merge_j].poi
nts,
            centroid = computeCentroid(data, clusters[merge_i].points
+
                                       clusters[merge_j].points
),
            height = min_distance,
            left_child = clusters[merge_i],
            right_child = clusters[merge_j]
        )

        // Записуємо об'єднання в дендрограму
        dendrogram.append({
            cluster1_id: clusters[merge_i].id,
            cluster2_id: clusters[merge_j].id,
            new_cluster_id: next_cluster_id,
            distance: min_distance
        })

        // Оновлюємо список кластерів
        clusters.remove(clusters[merge_j])
        clusters.remove(clusters[merge_i])
        clusters.append(new_cluster)

        // Оновлюємо матрицю відстаней
        updateDistanceMatrix(distance_matrix, merge_i, merge_j,
                             new_cluster, clusters, linkage_method)
```

```
        next_cluster_id += 1

    return HierarchicalModel(
        root = clusters[1],
        dendrogram = dendrogram,
        distance_matrix = distance_matrix
    )

function computeInitialDistanceMatrix(data: HierarchicalData): matrix
    n = data.n
    distance_matrix = matrix[n][n]

    for i = 1 to n do
        distance_matrix[i][i] = 0
        for j = i+1 to n do
            dist = euclideanDistance(data.X[i], data.X[j])
            distance_matrix[i][j] = dist
            distance_matrix[j][i] = dist

    return distance_matrix

function computeClusterDistance(cluster1: Cluster, cluster2: Cluster,
                                distance_matrix: matrix, method: strin
g): float
    if method == "single" then
        return singleLinkageDistance(cluster1, cluster2, distance_mat
rix)
    else if method == "complete" then
        return completeLinkageDistance(cluster1, cluster2, distance_m
atrix)
    else if method == "average" then
        return averageLinkageDistance(cluster1, cluster2, distance_ma
trix)
    else if method == "ward" then
        return wardLinkageDistance(cluster1, cluster2, distance_matri
x)
    else if method == "centroid" then
        return centroidLinkageDistance(cluster1, cluster2)

function singleLinkageDistance(cluster1: Cluster, cluster2: Cluster,
                               distance_matrix: matrix): float
    min_distance = infinity

    for i in cluster1.points do
        for j in cluster2.points do
```

```
            if distance_matrix[i][j] < min_distance then
                min_distance = distance_matrix[i][j]

    return min_distance

function completeLinkageDistance(cluster1: Cluster, cluster2: Cluster
,
                                 distance_matrix: matrix): float
    max_distance = -infinity

    for i in cluster1.points do
        for j in cluster2.points do
            if distance_matrix[i][j] > max_distance then
                max_distance = distance_matrix[i][j]

    return max_distance

function averageLinkageDistance(cluster1: Cluster, cluster2: Cluster,
                                distance_matrix: matrix): float
    sum_distance = 0.0
    count = 0

    for i in cluster1.points do
        for j in cluster2.points do
            sum_distance += distance_matrix[i][j]
            count += 1

    return sum_distance / count

function wardLinkageDistance(cluster1: Cluster, cluster2: Cluster,
                             distance_matrix: matrix): float
    n1 = length(cluster1.points)
    n2 = length(cluster2.points)

    centroid_distance = euclideanDistance(cluster1.centroid, cluster2
.centroid)

    return sqrt(n1 * n2 / (n1 + n2)) * centroid_distance
```

Формула Ланса-Вільямса

Для ефективного оновлення матриці відстаней використовується формула Ланса-Вільямса:

```
function updateDistanceMatrix(distance_matrix: matrix, i: integer, j:
integer,
                              new_cluster: Cluster, clusters: list, met
```

```
hod: string)
    // Параметри для різних методів зв'язку
    if method == "single" then
        alpha_i = 0.5
        alpha_j = 0.5
        beta = 0
        gamma = -0.5
    else if method == "complete" then
        alpha_i = 0.5
        alpha_j = 0.5
        beta = 0
        gamma = 0.5
    else if method == "average" then
        n_i = length(clusters[i].points)
        n_j = length(clusters[j].points)
        n_ij = n_i + n_j
        alpha_i = n_i / n_ij
        alpha_j = n_j / n_ij
        beta = 0
        gamma = 0
    else if method == "ward" then
        // Складніші формули для методу Варда
        // Обчислюються на основі розмірів кластерів

    // Оновлення відстаней до нового кластера
    for k = 1 to length(clusters) do
        if k != i and k != j then
            d_ki = distance_matrix[k][i]
            d_kj = distance_matrix[k][j]
            d_ij = distance_matrix[i][j]

            new_distance = alpha_i * d_ki + alpha_j * d_kj +
                           beta * d_ij + gamma * abs(d_ki - d_kj)

            // Оновлюємо матрицю для нового кластера
            // (тут потрібна додаткова логіка для правильного індексу
вання)
```

Дивізивна ієрархічна кластеризація

Дивізивний підхід працює в протилежному напрямку:

```
function divisiveClustering(data: HierarchicalData): HierarchicalMode
l
    // Початковий кластер містить всі точки
    root_cluster = Cluster(
        id = 1,
```

```
        points = range(1, data.n),
        centroid = computeCentroid(data, range(1, data.n)),
        height = 0,
        left_child = null,
        right_child = null
    )

    dendrogram = []
    next_cluster_id = 2

    // Черга кластерів для поділу
    queue = [root_cluster]

    while not queue.isEmpty() and shouldContinueSplitting(queue) do
        // Вибираємо кластер для поділу (наприклад, з максимальним ді
аметром)
        cluster_to_split = selectClusterToSplit(queue)
        queue.remove(cluster_to_split)

        if length(cluster_to_split.points) <= 1 then
            continue

        // Розділяємо кластер на два підкластери
        subcluster1, subcluster2 = splitCluster(data, cluster_to_spli
t)

        // Створюємо нові кластери
        left_cluster = Cluster(
            id = next_cluster_id,
            points = subcluster1,
            centroid = computeCentroid(data, subcluster1),
            height = cluster_to_split.height + computeSplitHeight(sub
cluster1, subcluster2),
            left_child = null,
            right_child = null
        )
        next_cluster_id += 1

        right_cluster = Cluster(
            id = next_cluster_id,
            points = subcluster2,
            centroid = computeCentroid(data, subcluster2),
            height = cluster_to_split.height + computeSplitHeight(sub
cluster1, subcluster2),
            left_child = null,
```

```
            right_child = null
        )
        next_cluster_id += 1

        // Оновлюємо батьківський кластер
        cluster_to_split.left_child = left_cluster
        cluster_to_split.right_child = right_cluster

        // Додаємо нові кластери до черги
        if length(left_cluster.points) > 1 then
            queue.append(left_cluster)
        if length(right_cluster.points) > 1 then
            queue.append(right_cluster)

        // Записуємо поділ у дендрограму
        dendrogram.append({
            parent_id: cluster_to_split.id,
            left_child_id: left_cluster.id,
            right_child_id: right_cluster.id,
            split_height: computeSplitHeight(subcluster1, subcluster2
)
        })

    return HierarchicalModel(root_cluster, dendrogram, null)

function splitCluster(data: HierarchicalData, cluster: Cluster): (lis
t, list)
    // Можна використовувати різні методи поділу
    // Наприклад, K-Means з K=2
    points = cluster.points
    cluster_data = extractPoints(data, points)

    kmeans_result = kMeans(cluster_data, K=2)

    subcluster1 = []
    subcluster2 = []

    for i = 1 to length(points) do
        if kmeans_result.labels[i] == 1 then
            subcluster1.append(points[i])
        else
            subcluster2.append(points[i])

    return (subcluster1, subcluster2)
```

Визначення оптимальної кількості кластерів

Після побудови повної ієрархії необхідно визначити оптимальний рівень зрізу дендрограми:

```
function findOptimalClusters(model: HierarchicalModel, max_k: integer
): integer
    silhouette_scores = []

    for k = 2 to max_k do
        clusters = cutDendrogram(model, k)
        score = computeSilhouetteScore(clusters)
        silhouette_scores.append(score)

    // Знаходимо k з максимальним silhouette score
    optimal_k = argmax(silhouette_scores) + 1

    return optimal_k

function cutDendrogram(model: HierarchicalModel, k: integer): list of
Cluster
    // Знаходимо висоту зрізу для отримання k кластерів
    if k >= model.n then
        return individualClusters(model)

    // Сортуємо об'єднання за висотою
    sorted_merges = sort(model.dendrogram, by="distance")

    // Висота зрізу - між (n-k) та (n-k+1) об'єднаннями
    cut_height = (sorted_merges[model.n - k].distance +
                  sorted_merges[model.n - k + 1].distance) / 2

    // Обходимо дерево та збираємо кластери
    clusters = []
    traverseAndCut(model.root, cut_height, clusters)

    return clusters

function traverseAndCut(node: Cluster, cut_height: float, clusters: l
ist)
    if node.height <= cut_height or node.left_child == null then
        // Це лист або вузол нижче висоти зрізу
        clusters.append(node)
    else
        // Рекурсивно обходимо дочірні вузли
        traverseAndCut(node.left_child, cut_height, clusters)
        traverseAndCut(node.right_child, cut_height, clusters)
```

Візуалізація дендрограми

```
function plotDendrogram(model: HierarchicalModel, labels: array of st
ring)
    // Обчислюємо координати для візуалізації
    coordinates = computeDendrogramCoordinates(model.root, 0, model.n
)

    // Рекурсивно малюємо гілки дендрограми
    drawBranches(model.root, coordinates)

    // Додаємо підписи листів
    for i = 1 to model.n do
        drawLabel(i, labels[i], coordinates[i].x, 0)

function computeDendrogramCoordinates(node: Cluster, left: float, rig
ht: float): Coordinates
    if node.left_child == null then
        // Листовий вузол
        x = (left + right) / 2
        y = 0
        return Coordinates(x, y)

    // Рекурсивно обчислюємо координати дочірніх вузлів
    mid = (left + right) / 2
    left_coords = computeDendrogramCoordinates(node.left_child, left,
mid)
    right_coords = computeDendrogramCoordinates(node.right_child, mid
, right)

    // Координати поточного вузла
    x = (left_coords.x + right_coords.x) / 2
    y = node.height

    return Coordinates(x, y)
```

Оптимізації для великих датасетів

1. SLINK (Single Linkage)

Алгоритм SLINK має складність $O(n^2)$ для single linkage:

```
function SLINK(data: HierarchicalData): HierarchicalModel
    n = data.n

    // Масиви для представлення дендрограми
    pi = array[n+1]  // вказівник на представника
    lambda = array[n+1]  // висота об'єднання
```

```
    // Ініціалізація
    for i = 1 to n do
        pi[i] = i
        lambda[i] = infinity

    // Основний цикл
    for i = 1 to n do
        for j = 1 to i-1 do
            d_ij = euclideanDistance(data.X[i], data.X[j])

            // Оновлення за алгоритмом SLINK
            if lambda[j] >= d_ij then
                // j приєднується до кластера i
                pi[j] = i
                lambda[j] = d_ij
            else
                // Знаходимо представника j
                rep_j = j
                while pi[rep_j] != rep_j and lambda[rep_j] >= d_ij do
                    rep_j = pi[rep_j]

                if lambda[rep_j] < d_ij then
                    pi[i] = rep_j
                else
                    pi[i] = i

    return constructDendrogramFromPointers(pi, lambda)
```

Паралельна ієрархічна кластеризація

```
function parallelHierarchicalClustering(data: HierarchicalData,
                                        num_threads: integer): Hierarc
hicalModel
    // Розподіляємо обчислення матриці відстаней
    n = data.n
    distance_matrix = matrix[n][n]

    parallel for thread_id = 1 to num_threads do
        // Кожен потік обчислює свою частину матриці
        start_row = (thread_id - 1) * n / num_threads + 1
        end_row = thread_id * n / num_threads

        for i = start_row to end_row do
            for j = i+1 to n do
                distance_matrix[i][j] = euclideanDistance(data.X[i],
```

```
data.X[j])
                distance_matrix[j][i] = distance_matrix[i][j]

    // Паралельний пошук мінімуму на кожній ітерації
    clusters = initializeClusters(data)

    while length(clusters) > 1 do
        // Паралельний пошук мінімальної відстані
        local_mins = array[num_threads]

        parallel for thread_id = 1 to num_threads do
            local_mins[thread_id] = findLocalMinimum(clusters, distan
ce_matrix,
                                                     thread_id, num_t
hreads)

        // Знаходимо глобальний мінімум
        global_min = findGlobalMinimum(local_mins)

        // Об'єднуємо кластери (послідовно для консистентності)
        mergeClusters(clusters, global_min.i, global_min.j)
        updateDistanceMatrixParallel(distance_matrix, global_min.i, g
lobal_min.j,
                                     clusters, num_threads)
```

Область застосування

Ієрархічна кластеризація широко використовується в різних галузях:

- Біоінформатика: Побудова філогенетичних дерев, класифікація генів та білків, аналіз експресії генів.

- Таксономія: Класифікація організмів, створення ієрархічних систем в біології.

- Маркетинг: Сегментація клієнтів з ієрархічною структурою, аналіз споживчої поведінки.

- Обробка документів: Організація документів у тематичні ієрархії, кластеризація новин.

- Соціальні мережі: Виявлення ієрархічних спільнот, аналіз структури мереж.

- Обробка зображень: Сегментація зображень, розпізнавання текстур з різними рівнями деталізації.

Приклади застосування

Приклад 1: Базова агломеративна кластеризація

Розглянемо простий приклад з 5 точками на площині:

```
Дані:
A = [1, 1]
B = [2, 1]
C = [4, 3]
D = [5, 4]
E = [6, 3]

Крок 1: Початкова матриця відстаней
     A     B     C     D     E
A   0     1    3.61  5.66  5.83
B   1     0    2.83  4.24  4.47
C  3.61  2.83   0    1.41  2.00
D  5.66  4.24  1.41   0    1.41
E  5.83  4.47  2.00  1.41   0

Використовуємо single linkage

Ітерація 1:
Мінімальна відстань: d(A,B) = 1
Об'єднуємо A та B в кластер AB
Оновлена матриця:
     AB    C     D     E
AB    0   2.83  4.24  4.47
C   2.83   0    1.41  2.00
D   4.24  1.41   0    1.41
E   4.47  2.00  1.41   0

Ітерація 2:
Мінімальна відстань: d(C,D) = d(D,E) = 1.41
Об'єднуємо C та D в кластер CD
     AB    CD    E
AB    0   2.83  4.47
CD  2.83   0    1.41
E   4.47  1.41   0

Ітерація 3:
Мінімальна відстань: d(CD,E) = 1.41
Об'єднуємо CD та E в кластер CDE
     AB   CDE
AB    0   2.83
CDE 2.83   0
```

```
Ітерація 4:
Об'єднуємо AB та CDE на відстані 2.83

Дендрограма:
Висота
2.83 |     ┌──────────────┐
     |     │              │
1.41 |     │      ┌───────┼───┐
     |     │      │       │   │
1.00 |  ┌──┼──┐   │       │   │
     |  │  │  │   │       │   │
0.00 | A  B  │   C       D   E
            │
         Кластери
```

Приклад 2: Класифікація видів ірисів

Застосування до класичного датасету Iris:

```
struct IrisData {
    sepal_length: float
    sepal_width: float
    petal_length: float
    petal_width: float
    species: string
}

function clusterIrisSpecies(iris_data: array of IrisData)
    // Підготовка даних
    X = extractFeatures(iris_data)
    X_normalized = standardize(X)

    // Ієрархічна кластеризація з методом Варда
    model = agglomerativeClustering(X_normalized, linkage_method="war
d")

    // Визначення оптимальної кількості кластерів
    optimal_k = 3  // Знаємо що є 3 види
    clusters = cutDendrogram(model, optimal_k)

    // Аналіз результатів
    for i = 1 to optimal_k do
        cluster_points = getClusterPoints(clusters[i])
        species_distribution = analyzeSpeciesInCluster(cluster_points
, iris_data)

        print("Кластер", i, ":")
```

```
        print("  Домінуючий вид:", getDominantSpecies(species_distrib
ution))
        print("  Чистота:", calculatePurity(species_distribution))
        print("  Середні характеристики:")
        print("    Sepal length:", mean(sepal_lengths_in_cluster))
        print("    Sepal width:", mean(sepal_widths_in_cluster))
        print("    Petal length:", mean(petal_lengths_in_cluster))
        print("    Petal width:", mean(petal_widths_in_cluster))

// Результат:
Кластер 1:
  Домінуючий вид: Iris-setosa
  Чистота: 100%
  Середні характеристики:
    Sepal length: 5.01
    Sepal width: 3.43
    Petal length: 1.46
    Petal width: 0.25

Кластер 2:
  Домінуючий вид: Iris-versicolor
  Чистота: 94%
  Середні характеристики:
    Sepal length: 5.94
    Sepal width: 2.77
    Petal length: 4.26
    Petal width: 1.33

Кластер 3:
  Домінуючий вид: Iris-virginica
  Чистота: 92%
```

Приклад 3: Аналіз генетичних даних

Побудова філогенетичного дерева:

```
struct GeneticSequence {
    organism_name: string
    dna_sequence: string
}

function buildPhylogeneticTree(sequences: array of GeneticSequence)
    // Обчислення матриці генетичних відстаней
    n = length(sequences)
    distance_matrix = matrix[n][n]

    for i = 1 to n do
```

```
        for j = i+1 to n do
            // Використовуємо відстань Хеммінга для ДНК послідовносте
й
            distance = hammingDistance(sequences[i].dna_sequence,
                                       sequences[j].dna_sequence)
            distance_matrix[i][j] = distance
            distance_matrix[j][i] = distance

    // UPGMA (Unweighted Pair Group Method with Arithmetic Mean)
    model = upgmaClustering(distance_matrix, sequences)

    // Калібрування дерева за молекулярним годинником
    calibrated_tree = calibrateWithMolecularClock(model, known_diverg
ence_times)

    return calibrated_tree

function upgmaClustering(distance_matrix: matrix, sequences: array):
PhylogeneticTree
    n = length(sequences)
    clusters = initializeClustersWithSequences(sequences)
    cluster_sizes = array[n]

    for i = 1 to n do
        cluster_sizes[i] = 1

    while length(clusters) > 1 do
        // Знаходимо найближчі кластери
        min_i, min_j, min_distance = findMinimumDistance(distance_mat
rix)

        // Створюємо новий вузол дерева
        new_node = PhylogeneticNode(
            left_child = clusters[min_i],
            right_child = clusters[min_j],
            branch_length = min_distance / 2,
            divergence_time = estimateDivergenceTime(min_distance)
        )

        // Оновлюємо матрицю відстаней (UPGMA формула)
        new_distances = []
        for k = 1 to length(clusters) do
            if k != min_i and k != min_j then
                d_ik = distance_matrix[min_i][k]
                d_jk = distance_matrix[min_j][k]
```

```
                new_distance = (cluster_sizes[min_i] * d_ik +
                                cluster_sizes[min_j] * d_jk) /
                                (cluster_sizes[min_i] + cluster_sizes[
min_j])

                new_distances.append(new_distance)

        // Оновлюємо структури даних
        updateClustersAndMatrix(clusters, distance_matrix, cluster_si
zes,
                                min_i, min_j, new_node, new_distances)

    return PhylogeneticTree(root = clusters[1])
```

Приклад 4: Сегментація клієнтів в e-commerce

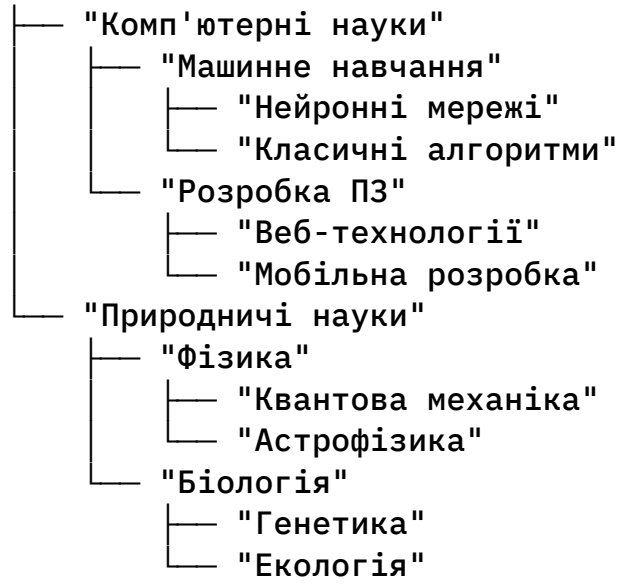
```
struct CustomerProfile {
    customer_id: integer
    total_purchases: float
    avg_order_value: float
    purchase_frequency: float
    days_since_last_purchase: integer
    product_categories: array
    preferred_payment: string
}

function hierarchicalCustomerSegmentation(customers: array of Custome
rProfile)
    // Підготовка ознак
    features = []
    for customer in customers do
        feature_vector = [
            log(customer.total_purchases + 1),
            customer.avg_order_value / 100,
            customer.purchase_frequency,
            1 / (customer.days_since_last_purchase + 1),
            computeCategoryDiversity(customer.product_categories),
            isDigitalPayment(customer.preferred_payment) ? 1 : 0
        ]
        features.append(normalize(feature_vector))

    // Ієрархічна кластеризація
    model = agglomerativeClustering(features, linkage_method="complet
e")
```

```
    // Визначаємо різні рівні сегментації
    macro_segments = cutDendrogram(model, k=3)  // Високорівневі сегм
енти
    micro_segments = cutDendrogram(model, k=10) // Детальні сегменти

    // Аналіз ієрархії сегментів
    segment_hierarchy = buildSegmentHierarchy(model, customers)

    // Характеристика кожного рівня
    for level in ["macro", "micro"] do
        segments = level == "macro" ? macro_segments : micro_segments

        for segment in segments do
            profile = analyzeSegmentProfile(segment, customers)

            print("Сегмент", segment.id, "(Рівень:", level, ")")
            print("  Розмір:", length(segment.customers))
            print("  Середня вартість клієнта:", profile.avg_customer
_value)
            print("  Основні категорії:", profile.top_categories)
            print("  Рекомендована стратегія:", suggestStrategy(profi
le))

// Приклад результату:
Макро-сегмент 1: "VIP клієнти"
  Розмір: 1,200
  CLV: $5,420
  Стратегія: Персоналізовані пропозиції, ексклюзивні знижки

  Мікро-сегмент 1.1: "Техно-ентузіасти"
    Розмір: 450
    Основні категорії: Електроніка, Гаджети

  Мікро-сегмент 1.2: "Модні покупці"
    Розмір: 750
    Основні категорії: Одяг, Аксесуари
```

Приклад 5: Аналіз тексту - ієрархічна організація документів

```
struct Document {
    id: string
    title: string
    content: string
    tf_idf_vector: vector
}
```

```
function hierarchicalDocumentClustering(documents: array of Document)
    // Обчислення косинусної подібності між документами
    n = length(documents)
    similarity_matrix = matrix[n][n]

    for i = 1 to n do
        for j = i+1 to n do
            similarity = cosineSimilarity(documents[i].tf_idf_vector,
                                          documents[j].tf_idf_vector)
            // Конвертуємо подібність у відстань
            distance = 1 - similarity
            similarity_matrix[i][j] = distance
            similarity_matrix[j][i] = distance

    // Ієрархічна кластеризація з average linkage
    model = agglomerativeClustering(similarity_matrix,
                                    linkage_method="average",
                                    precomputed=true)

    // Створення тематичної ієрархії
    topic_hierarchy = extractTopicHierarchy(model, documents)

    // Функція для витягування тем на різних рівнях
    function extractTopicsAtLevel(node: Cluster, level: integer): Top
ics
        if level == 0 or node.isLeaf() then
            return extractKeywords(node.documents)

        // Рекурсивно обходимо дерево
        left_topics = extractTopicsAtLevel(node.left_child, level - 1
)
        right_topics = extractTopicsAtLevel(node.right_child, level -
1)

        // Об'єднуємо та узагальнюємо теми
        combined_topics = mergeTopics(left_topics, right_topics)
        generalized_topics = generalizeTopics(combined_topics)

        return generalized_topics

    // Візуалізація тематичної ієрархії
    printTopicHierarchy(topic_hierarchy)

// Приклад результату:
Корінь: "Технології та наука"
```



Переваги та недоліки

Переваги

- Не потребує вказання кількості кластерів: Будує повну ієрархію, дозволяючи вибрати оптимальний рівень деталізації.

- Детермінованість: Для фіксованого методу зв’язку завжди дає однаковий результат.

- Інтерпретованість: Дендрограма надає інтуїтивне візуальне представлення структури даних.

- Гнучкість: Різні методи зв’язку дозволяють знаходити кластери різної форми.

- Ієрархічна структура: Природно моделює вкладені відношення в даних.

- Працює з матрицею відстаней: Може використовувати будь-яку міру подібності/відмінності.

Недоліки

- Обчислювальна складність: $O(n^3)$ для наївної реалізації, $O(n^2)$ для оптимізованих версій.

- Використання пам’яті: Потребує зберігання матриці відстаней $O(n^2)$.

- Незворотність: Неможливо виправити помилкові об’єднання на ранніх етапах.

- Чутливість до викидів: Особливо для single та complete linkage.

- Складність вибору рівня зрізу: Не завжди очевидно, де розрізати дендрограму.

- Погано масштабується: Непрактично для дуже великих датасетів (>10,000 точок).

Контрольні запитання

1. Яка різниця між агломеративним та дивізивним підходами в ієрархічній кластеризації?

2. Поясніть різницю між single, complete та average linkage методами.

3. Що таке дендрограма та як її інтерпретувати?

4. Яка часова та просторова складність агломеративної кластеризації?

5. Чому метод Варда мінімізує внутрішньокластерну дисперсію?

6. Як формула Ланса-Вільямса спрощує оновлення матриці відстаней?

7. Які методи використовуються для визначення оптимальної кількості кластерів?

8. Чим UPGMA відрізняється від інших методів зв’язку?

9. Які переваги має ієрархічна кластеризація над K-Means?

10. Як можна прискорити ієрархічну кластеризацію для великих датасетів?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритмів ієрархічної кластеризації

2. Мета

Вивчити принципи ієрархічної кластеризації, реалізувати агломеративний алгоритм з різними методами зв’язку, навчитися будувати та інтерпретувати дендрограми, порівняти з іншими методами кластеризації.

3. Завдання

Реалізувати агломеративну кластеризацію з підтримкою різних методів зв’язку, створити функції для побудови та візуалізації дендрограм, реалізувати методи визначення оптимального рівня зрізу, додати оптимізації для великих датасетів.

4. Опис

Лабораторна робота охоплює повний цикл ієрархічного кластерного аналізу: від обчислення матриці відстаней до інтерпретації результатів. Студенти реалізують різні варіанти алгоритму та досліджують їх властивості.

5. Порядок виконання

- Реалізувати обчислення матриці відстаней для різних метрик
- Створити базовий агломеративний алгоритм
- Додати підтримку методів single, complete, average linkage
- Реалізувати метод Варда
- Створити функцію побудови дендрограми
- Реалізувати візуалізацію дендрограми
- Додати методи для зрізу на різних рівнях
- Реалізувати обчислення cophenetic correlation
- Створити оптимізовану версію (SLINK або інша)
- Провести експерименти та порівняння

6. Результат

Повнофункціональна бібліотека ієрархічної кластеризації з візуалізацією, підтримкою різних методів зв'язку та метрик, інструменти для аналізу якості кластеризації, порівняльний аналіз з іншими методами.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас HierarchicalClustering з методами fit, fit_predict
- Реалізація різних методів зв'язку
- Функції побудови та візуалізації дендрограми
- Методи для оптимального зрізу
- Обчислення метрик якості

2. Продемонструвати роботу додатку:

- Кластеризація різних типів даних
- Візуалізація дендрограм
- Порівняння методів зв'язку
- Аналіз стабільності результатів
- Застосування до реальних даних

3. Бути готовим відповісти на питання:

- Коли який метод зв'язку використовувати?

- Як інтерпретувати висоту в дендрограмі?
- Що показує cophenetic correlation?
- Як масштабувати для великих даних?

Тести

1. Агломеративна кластеризація починає з:
- 1) Одного кластера
- 2) K кластерів
- 3) N кластерів (кожна точка - окремий кластер)
- 4) Випадкових кластерів

2. Single linkage схильний створювати:
- 1) Компактні кластери
- 2) Витягнуті кластери (chaining effect)
- 3) Сферичні кластери
- 4) Рівномірні кластери

3. Метод Варда мінімізує:
- 1) Міжкластерну відстань
- 2) Внутрішньокластерну дисперсію
- 3) Кількість кластерів
- 4) Висоту дендрограми

4. Складність наївної агломеративної кластеризації:
- 1) O(n)
- 2) O(n log n)
- 3) $O(n^2)$
- 4) $O(n^3)$

5. Cophenetic correlation вимірює:
- 1) Якість кластерів
- 2) Відповідність дендрограми вихідним відстаням
- 3) Кількість кластерів
- 4) Швидкість алгоритму

Відповіді:

1-3,

2-2,

3-2,

4-4,

5-2

Тести підвищеної складності

1. Чому complete linkage стійкий до викидів?

- 1) Ігнорує викиди
- 2) Використовує максимальну відстань
- 3) Використовує медіану
- 4) Видаляє викиди

2. UPGMA припускає:

- 1) Нормальний розподіл
- 2) Постійну швидкість еволюції (molecular clock)
- 3) Бінарні дані
- 4) Лінійні залежності

3. Lance-Williams формула дозволяє:

- 1) Обчислювати початкові відстані
- 2) Ефективно оновлювати матрицю відстаней
- 3) Будувати дендрограму
- 4) Вибирати кількість кластерів

4. Дивізивна кластеризація зазвичай:

- 1) Швидша за агломеративну
- 2) Повільніша за агломеративну
- 3) Однакова за швидкістю
- 4) Не порівнянна

5. Centroid linkage може мати:

- 1) Інверсії в дендрограмі
- 2) Негативні відстані
- 3) Цикли

- 4) Паралельні гілки

6. SLINK алгоритм оптимізований для:

- 1) Complete linkage
- 2) Single linkage
- 3) Average linkage
- 4) Ward’s method

7. Для текстових даних краще використовувати:

- 1) Евклідову відстань
- 2) Косинусну відстань
- 3) Манхеттенську відстань
- 4) Чебишева відстань

8. Ultrametric property означає:

- 1) $d(x,z) \leqslant \max(d(x,y), d(y,z))$
- 2) $d(x,y) = d(y,x)$
- 3) $d(x,x) = 0$
- 4) $d(x,y) \geqslant 0$

9. Які методи зв’язку гарантують монотонність?

- 1) Single та complete
- 2) Тільки Ward
- 3) Centroid та median
- 4) Всі методи

10. Для виявлення ієрархічних спільнот в графах використовують:

- 1) Модулярність на різних рівнях
- 2) Тільки відстань
- 3) Кількість ребер
- 4) Ступені вершин

Відповіді:

1. 2) Максимальна відстань робить метод консервативним
2. 2) Основне припущення для філогенетичних дерев
3. 2) Рекурсивна формула для ефективного оновлення

4. 2) Експоненційна складність повного перебору поділів
5. 1) Центроїди можуть зближуватися при об’єднанні
6. 2) $O(n^2)$ алгоритм спеціально для single linkage
7. 2) Нормалізована міра для високорозмірних розріджених векторів
8. 1) Сильніша властивість ніж трикутна нерівність
9. 1) Гарантують неспадання відстаней при об’єднанні
10. 1) Ієрархічна оптимізація модулярності

Завдання для самостійної роботи

1. Реалізуйте дивізивний алгоритм з використанням K-Means для поділу.
2. Створіть інтерактивну візуалізацію дендрограми з zoom та деталями.
3. Розробіть метод автоматичного визначення оптимального зрізу.
4. Реалізуйте fuzzy hierarchical clustering.
5. Створіть паралельну версію для багатоядерних систем.
6. Розробіть інкрементальну ієрархічну кластеризацію.
7. Реалізуйте constrained hierarchical clustering.
8. Створіть гібрид з DBSCAN для різної щільності.
9. Розробіть метод для категоріальних даних.
10. Дослідіть застосування для deep learning embeddings.

Теми лабораторних робіт

1. Базова реалізація агломеративної кластеризації.
2. Порівняння методів зв’язку на різних датасетах.
3. Побудова філогенетичних дерев.
4. Ієрархічна сегментація зображень.
5. Кластеризація часових рядів.
6. Аналіз соціальних мереж через ієрархії.
7. Організація документів в тематичні ієрархії.
8. Оптимізація для великих даних (BIRCH).
9. Комбінування з іншими методами ML.
10. Застосування для аналізу геномних даних.

15.3. Класифікація

У цьому розділі ми розглянемо ключові методи класифікації — задачі, в якій метою є віднесення вхідного об’єкта до однієї з наперед заданих категорій. Алгоритми класифікації широко застосовуються в різних галузях: від діагностики захворювань і виявлення спаму до рекомендаційних систем і прогнозування кредитної безпеки. Кожен із описаних інструментів має свої сильні сторони й обмеження, тож вибір оптимального підходу залежить від природи даних, вимог до інтерпретованості моделі та доступних обчислювальних ресурсів.

Дерево рішень (Decision Tree) будує ієрархію «якщо-то» правил, що дає зрозумілу й візуалізовану модель, проте може надмірно «підганятися» під навчальні дані. Random Forest усуває цю вразливість, поєднуючи багато дерев у ансамбль через вагове голосування, що підвищує стійкість та точність. Gradient Boosting глибше «качнув» ідею ансамблю: кожне наступне дерево коригує помилки попередніх, що дозволяє досягти ще вищої якості класифікації, але потребує ретельного налаштування гіперпараметрів.

Метод опорних векторів (SVM) шукає оптимальну гіперплощину, яка максимально відокремлює класи в просторах високої вимірності, й добре працює на розріджених чи нерівноважених даних. Наївний байєсівський класифікатор (Bayes) спирається на байєсівську ймовірнісну модель та допущення незалежності ознак: незважаючи на його простоту, він часто демонструє високу швидкодію та конкурентну точність, особливо в задачах текстової класифікації. У наступних підрозділах ми детально розглянемо кожен із цих методів, їх математичні основи та практичні приклади застосування.

Класифікація: дерево рішень

Теоретичні відомості

Дерево рішень є одним з найпопулярніших алгоритмів машинного навчання для задач класифікації та регресії. Цей алгоритм створює модель у вигляді деревоподібної структури, де кожен внутрішній вузол представляє тест на певній ознаці, кожна гілка відповідає результату тесту, а кожен листковий вузол містить мітку класу або прогнозоване значення.

Основна ідея алгоритму полягає в рекурсивному розбитті простору ознак на прямокутні області таким чином, щоб максимізувати однорідність міток класів у кожній області. Процес побудови дерева починається з кореневого вузла, який містить всі навчальні приклади, і продовжується шляхом вибору найкращої ознаки для розщеплення на кожному кроці.

Ключові концепції

Ентропія - це міра невизначеності або неоднорідності набору даних. Для набору S з прикладами, що належать до різних класів, ентропія обчислюється за формулою:

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$$

де p_i - частка прикладів, що належать до класу i, а c - загальна кількість класів. Ентропія досягає максимуму, коли всі класи представлені рівномірно, і дорівнює нулю, коли всі приклади належать до одного класу.

Приріст інформації (Information Gain) вимірює зменшення ентропії після розщеплення набору даних за певною ознакою. Для ознаки A приріст інформації обчислюється як:

$$IG(S, A) = H(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} H(S_v)$$

де $Values(A)$ - множина можливих значень ознаки A, S_v - підмножина прикладів з S, для яких ознака A має значення v, $|S|$ - кількість прикладів у множині S.

Індекс Джині - альтернативна міра неоднорідності, яка часто використовується замість ентропії:

$$Gini(S) = 1 - \sum_{i=1}^{c} p_i^2$$

Індекс Джині досягає мінімуму (0), коли всі приклади належать до одного класу, і максимуму, коли класи розподілені рівномірно.

Алгоритм побудови дерева рішень

Розглянемо алгоритм побудови дерева рішень для задачі класифікації. Алгоритм працює рекурсивно, починаючи з кореневого вузла і створюючи дочірні вузли шляхом розщеплення даних за найкращою ознакою на кожному кроці.

Основні кроки алгоритму:

1. Перевірка критеріїв зупинки. Якщо всі приклади належать до одного класу, створюється листковий вузол з відповідною міткою класу. Якщо досягнута максимальна глибина дерева або кількість прикладів менша за мінімальний поріг, створюється листковий вузол з найпоширенішим класом.

2. Вибір найкращої ознаки для розщеплення. Для кожної доступної ознаки обчислюється критерій розщеплення (приріст інформації або зменшення індексу Джині). Вибирається ознака з найкращим значенням критерію.

3. Створення внутрішнього вузла з обраною ознакою та розщеплення даних відповідно до значень цієї ознаки.

4. Рекурсивна побудова піддерев для кожної підмножини даних, отриманої після розщеплення.

Псевдокод алгоритму ID3

```
функція ПобудуватиДерево(приклади, ознаки, глибина)
    // Перевірка критеріїв зупинки
    якщо всі приклади належать до одного класу C тоді
        повернути листковий вузол з міткою C

    якщо ознаки порожні або глибина = максимальна_глибина тоді
        C = найпоширеніший клас у прикладах
        повернути листковий вузол з міткою C

    // Вибір найкращої ознаки
    найкраща_ознака = null
    максимальний_приріст = -∞

    для кожної ознаки A з ознак виконати
        приріст = ОбчислитиПрирістІнформації(приклади, A)
        якщо приріст > максимальний_приріст тоді
            максимальний_приріст = приріст
            найкраща_ознака = A

    // Створення внутрішнього вузла
    вузол = новий внутрішній вузол з ознакою найкраща_ознака

    // Розщеплення даних і рекурсивна побудова піддерев
    для кожного значення v ознаки найкраща_ознака виконати
        приклади_v = підмножина прикладів де найкраща_ознака = v
```

```
        якщо приклади_v не порожні тоді
            нові_ознаки = ознаки \ {найкраща_ознака}
            піддерево = ПобудуватиДерево(приклади_v, нові_ознаки, гли
бина + 1)
            додати гілку від вузол до піддерево з міткою v
        інакше
            C = найпоширеніший клас у прикладах
            додати листковий вузол з міткою C як дочірній до вузол

    повернути вузол

функція ОбчислитиПрирістІнформації(приклади, ознака)
    ентропія_до = ОбчислитиЕнтропію(приклади)

    ентропія_після = 0
    для кожного значення v ознаки виконати
        приклади_v = підмножина прикладів де ознака = v
        вага = |приклади_v| / |приклади|
        ентропія_після += вага * ОбчислитиЕнтропію(приклади_v)

    повернути ентропія_до - ентропія_після

функція ОбчислитиЕнтропію(приклади)
    якщо приклади порожні тоді повернути 0

    ентропія = 0
    для кожного класу c виконати
        p = частка прикладів класу c у приклади
        якщо p > 0 тоді
            ентропія -= p * log2(p)

    повернути ентропія
```

Алгоритм класифікації нового прикладу

Після побудови дерева рішень, класифікація нового прикладу виконується шляхом проходження від кореня до листка відповідно до значень ознак прикладу.

```
функція Класифікувати(дерево, приклад)
    поточний_вузол = дерево

    поки поточний_вузол не є листком виконати
        ознака = поточний_вузол.ознака
        значення = приклад[ознака]
```

```
        якщо існує гілка для значення тоді
            поточний_вузол = дочірній вузол по гілці значення
        інакше
            // Обробка невідомого значення
            поточний_вузол = найпоширеніший дочірній вузол

    повернути поточний_вузол.клас
```

Обрізання дерева (Pruning)

Обрізання дерева є важливою технікою для запобігання перенавчанню. Існують два основні підходи:

- Попереднє обрізання (Pre-pruning) - зупинка росту дерева під час його побудови при досягненні певних критеріїв: - Мінімальна кількість прикладів у вузлі - Максимальна глибина дерева - Мінімальний приріст інформації

- Наступне обрізання (Post-pruning) - видалення гілок з повністю побудованого дерева:

```
функція ОбрізатиДерево(дерево, валідаційні_дані)
    якщо дерево є листком тоді
        повернути дерево

    // Рекурсивно обрізати піддерева
    для кожного дочірнього вузла виконати
        дочірній = ОбрізатиДерево(дочірній, валідаційні_дані)

    // Оцінити точність з поточним піддеревом
    точність_з_піддеревом = ОцінитиТочність(дерево, валідаційні_дані)

    // Створити листок з найпоширенішим класом
    листок = новий листковий вузол з найпоширенішим класом
    точність_з_листком = ОцінитиТочність(листок, валідаційні_дані)

    якщо точність_з_листком >= точність_з_піддеревом тоді
        повернути листок
    інакше
        повернути дерево
```

Приклад застосування

Розглянемо приклад побудови дерева рішень для класифікації погоди для гри в теніс. Маємо навчальний набір даних з чотирма ознаками та бінарною класифікацією:

Навчальні дані:

- Прогноз: Сонячно, Хмарно, Дощ

- Температура: Спекотно, Помірно, Прохолодно

- Вологість: Висока, Нормальна

- Вітер: Слабкий, Сильний

- Клас: Грати, Не грати

Припустимо, у нас є 14 прикладів:

1. (Сонячно, Спекотно, Висока, Слабкий) → Не грати
2. (Сонячно, Спекотно, Висока, Сильний) → Не грати
3. (Хмарно, Спекотно, Висока, Слабкий) → Грати
4. (Дощ, Помірно, Висока, Слабкий) → Грати
5. (Дощ, Прохолодно, Нормальна, Слабкий) → Грати
6. (Дощ, Прохолодно, Нормальна, Сильний) → Не грати
7. (Хмарно, Прохолодно, Нормальна, Сильний) → Грати
8. (Сонячно, Помірно, Висока, Слабкий) → Не грати
9. (Сонячно, Прохолодно, Нормальна, Слабкий) → Грати
10. (Дощ, Помірно, Нормальна, Слабкий) → Грати
11. (Сонячно, Помірно, Нормальна, Сильний) → Грати
12. (Хмарно, Помірно, Висока, Сильний) → Грати
13. (Хмарно, Спекотно, Нормальна, Слабкий) → Грати
14. (Дощ, Помірно, Висока, Сильний) → Не грати

Крок 1: Обчислення початкової ентропії

З 14 прикладів: 9 - “Грати”, 5 - “Не грати”

$$H(S) = -\frac{9}{14}\log_2(\frac{9}{14}) - \frac{5}{14}\log_2(\frac{5}{14}) \approx 0.940$$

Крок 2: Обчислення приросту інформації для кожної ознаки

Для ознаки “Прогноз”: - Сонячно: 2 Грати, 3 Не грати - Хмарно: 4 Грати, 0 Не грати

- Дощ: 3 Грати, 2 Не грати

$$H(\text{Сонячно}) = -\frac{2}{5}\log_2(\frac{2}{5}) - \frac{3}{5}\log_2(\frac{3}{5}) \approx 0.971$$

$$H(\text{Хмарно}) = 0$$

(всі приклади одного класу)

$$H(\text{Дощ}) = -\frac{3}{5}\log_2(\frac{3}{5}) - \frac{2}{5}\log_2(\frac{2}{5}) \approx 0.971$$

$$IG(S, \text{Прогноз}) = 0.940 - [\frac{5}{14} \times 0.971 + \frac{4}{14} \times 0 + \frac{5}{14} \times 0.971] \approx 0.247$$

Аналогічно обчислюємо для інших ознак і отримуємо, що найбільший приріст інформації дає ознака “Прогноз”.

Крок 3: Створення кореневого вузла

Кореневий вузол містить тест на ознаці “Прогноз” з трьома гілками.

Крок 4: Рекурсивна побудова піддерев

Для гілки “Хмарно” всі приклади належать до класу “Грати”, тому створюємо листковий вузол.

Для гілок “Сонячно” та “Дощ” продовжуємо рекурсивну побудову, вибираючи найкращі ознаки для розщеплення відповідних підмножин даних.

Область застосування

Дерева рішень широко застосовуються в різних галузях завдяки своїй інтерпретованості та ефективності:

Медична діагностика - дерева рішень використовуються для створення експертних систем, що допомагають лікарям діагностувати захворювання на основі симптомів та результатів аналізів. Прозорість моделі дозволяє медичним працівникам зрозуміти логіку прийняття рішень.

Фінансовий сектор - банки та кредитні організації використовують дерева рішень для оцінки кредитоспроможності клієнтів, виявлення шахрайства та прийняття рішень щодо видачі кредитів. Модель може враховувати такі фактори, як кредитна історія, дохід, вік та інші демографічні показники.

Маркетинг та продажі - компанії застосовують дерева рішень для сегментації клієнтів, прогнозування відтоку клієнтів та персоналізації маркетингових кампаній. Алгоритм допомагає визначити, які характеристики клієнтів найбільше впливають на їхню поведінку.

Промисловість - дерева рішень використовуються для діагностики несправностей обладнання, оптимізації виробничих процесів та контролю якості продукції. Модель може аналізувати показники датчиків та історичні дані для прогнозування відмов.

Екологія та сільське господарство - алгоритм застосовується для класифікації типів ґрунтів, прогнозування врожайності, визначення оптимальних умов вирощування рослин та моніторингу стану довкілля.

Приклади застосування

Приклад 1: Діагностика захворювань серця

Медичний центр використовує дерево рішень для попередньої діагностики серцевих захворювань. Модель аналізує такі ознаки: - Вік пацієнта - Рівень холестерину - Артеріальний тиск - Наявність болю в грудях - Результати ЕКГ

Дерево рішень створює зрозумілі правила, наприклад: “Якщо вік > 50 І рівень холестерину > 240 І є біль у грудях, то високий ризик серцевого захворювання”. Це дозволяє лікарям швидко оцінити ризик та призначити додаткові обстеження.

Приклад 2: Прогнозування відтоку клієнтів телекомунікаційної компанії

Телекомунікаційна компанія використовує дерево рішень для визначення клієнтів з високим ризиком переходу до конкурентів. Модель аналізує: - Тривалість користування послугами - Кількість дзвінків до служби підтримки - Тарифний план - Середній рахунок за місяць - Використання додаткових послуг

На основі побудованого дерева компанія розробляє персоналізовані пропозиції для утримання клієнтів з високим ризиком відтоку.

Приклад 3: Автоматизація процесу схвалення кредитів

Банк впроваджує систему на основі дерева рішень для автоматизації першого етапу розгляду кредитних заявок. Система аналізує: - Кредитну історію заявника - Співвідношення доходу до запитуваної суми кредиту - Стаж роботи на поточному місці - Наявність застави - Вік заявника

Дерево рішень дозволяє автоматично схвалювати заявки з низьким ризиком та направляти складні випадки на додатковий розгляд кредитними експертами.

Переваги та недоліки

Переваги

Інтерпретованість - дерева рішень легко візуалізувати та інтерпретувати. Кожен шлях від кореня до листка представляє зрозуміле правило класифікації, що робить модель прозорою для користувачів без технічної освіти.

Відсутність необхідності в попередній обробці даних - алгоритм може працювати як з числовими, так і з категоріальними ознаками без необхідності нормалізації або стандартизації. Це спрощує підготовку даних та зменшує ймовірність помилок.

Обробка пропущених значень - дерева рішень можуть ефективно працювати з неповними даними, використовуючи різні стратегії обробки пропущених значень, такі як використання суррогатних розщеплень.

Нелінійні залежності - алгоритм здатний моделювати складні нелінійні зв'язки між ознаками та цільовою змінною без явного визначення форми залежності.

Швидкість прогнозування - після побудови дерева класифікація нових прикладів виконується дуже швидко, оскільки вимагає лише проходження шляху від кореня до листка.

Автоматичний відбір ознак - в процесі побудови дерева автоматично визначаються найважливіші ознаки, що може бути корисним для розуміння даних та зменшення розмірності.

Недоліки

Схильність до перенавчання - дерева рішень можуть створювати надто складні моделі, що ідеально підлаштовуються під навчальні дані, але погано узагальнюються на нові приклади. Це особливо проблематично для невеликих наборів даних.

Нестабільність - невеликі зміни в навчальних даних можуть призвести до побудови суттєво різних дерев. Ця чутливість ускладнює відтворюваність результатів.

Складність моделювання лінійних залежностей - для представлення простих лінійних зв'язків дерева рішень можуть створювати надмірно складні структури з багатьма розщепленнями.

Зміщення в бік ознак з великою кількістю рівнів - при використанні критерію приросту інформації алгоритм може надавати перевагу ознакам з більшою кількістю можливих значень, що не завжди відповідає їх реальній важливості.

Проблеми з незбалансованими класами - коли один клас значно переважає інші, дерево рішень може створювати зміщені моделі, що погано класифікують рідкісні класи.

Труднощі з екстраполяцією - дерева рішень не можуть робити прогнози за межами діапазону значень, представлених у навчальних даних, що обмежує їх застосування для прогнозування трендів.

Контрольні запитання

1. Що таке ентропія в контексті дерев рішень та як вона використовується для вибору найкращої ознаки для розщеплення?

2. Поясніть різницю між приростом інформації та індексом Джині як критеріями розщеплення в деревах рішень.

3. Чому дерева рішень схильні до перенавчання та які методи використовуються для вирішення цієї проблеми?

4. Як дерево рішень обробляє категоріальні ознаки з великою кількістю можливих значень?

5. Опишіть процес класифікації нового прикладу за допомогою побудованого дерева рішень.

6. Які переваги має попереднє обрізання порівняно з наступним обрізанням дерев рішень?

7. Чому дерева рішень вважаються інтерпретованими моделями машинного навчання?

8. Як обчислюється важливість ознак в алгоритмі дерева рішень?

9. Поясніть, чому дерева рішень можуть бути нестабільними та як це впливає на їх практичне застосування.

10. Які модифікації базового алгоритму ID3 існують та які проблеми вони вирішують?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму дерева рішень для задачі класифікації

2. Мета

Навчитися реалізовувати алгоритм побудови дерева рішень, розуміти принципи вибору оптимальних розщеплень та застосовувати методи запобігання перенавчанню.

3. Завдання

- Реалізувати алгоритм ID3 для побудови дерева рішень

- Додати підтримку різних критеріїв розщеплення (ентропія, індекс Джині)

- Реалізувати методи обрізання дерева

- Візуалізувати побудоване дерево

- Провести експерименти на різних наборах даних

4. Опис

Студенти повинні створити повну реалізацію алгоритму дерева рішень, включаючи структури даних для представлення дерева, функції обчислення критеріїв розщеплення, алгоритм рекурсивної побудови дерева та методи класифікації. Програма повинна підтримувати як категоріальні, так і числові ознаки, обробляти пропущені значення та включати механізми запобігання перенавчанню.

5. Порядок виконання

1. Створити структури даних для представлення вузлів дерева та навчальних даних

2. Реалізувати функції обчислення ентропії та приросту інформації

3. Реалізувати рекурсивний алгоритм побудови дерева

4. Додати підтримку числових ознак через пошук оптимальних порогів

5. Реалізувати алгоритм класифікації нових прикладів

6. Додати методи попереднього та наступного обрізання

7. Створити функцію візуалізації дерева

8. Провести тестування на стандартних наборах даних

6. Результат

Програма, що реалізує повнофункціональний алгоритм дерева рішень з можливістю навчання на різних наборах даних, класифікації нових прикладів та візуалізації побудованої моделі.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас або структура для представлення вузлів дерева

- Функції обчислення ентропії та приросту інформації

- Рекурсивна функція побудови дерева

- Функція класифікації нових прикладів

- Методи обрізання дерева

2. Продемонструвати роботу додатку:

- Завантаження та попередня обробка навчальних даних
- Побудова дерева з різними параметрами
- Візуалізація структури дерева
- Класифікація тестових прикладів
- Оцінка точності моделі

3. Бути готовим відповісти на питання:

- Як вибирається оптимальний поріг для числових ознак?
- Як обробляються пропущені значення в реалізації?
- Який вплив мають параметри обрізання на складність дерева?
- Як оцінюється важливість ознак у побудованій моделі?

Тести

1. Основна мета використання ентропії в алгоритмі дерева рішень:

1) Вимірювання складності дерева
2) Вимірювання неоднорідності набору даних
3) Обчислення відстані між прикладами
4) Нормалізація значень ознак

2. Приріст інформації для ознаки обчислюється як:

1) Сума ентропій всіх підмножин
2) Різниця між максимальною та мінімальною ентропією
3) Різниця між ентропією до та після розщеплення
4) Добуток ентропій підмножин

3. Індекс Джині досягає мінімального значення, коли:

1) Класи розподілені рівномірно
2) Всі приклади належать до одного класу
3) Кількість класів максимальна
4) Дані містять пропущені значення

4. Попереднє обрізання дерева рішень виконується:

1) Після повної побудови дерева
2) Під час побудови дерева

3) Перед початком навчання

4) Тільки для числових ознак

5. Основний недолік дерев рішень:

1) Повільна швидкість класифікації

2) Неможливість роботи з категоріальними даними

3) Схильність до перенавчання

4) Вимога нормалізації даних

Відповіді:

1-2,

2-3,

3-2,

4-2,

5-3

Тести підвищеної складності

1. При використанні дерева рішень для задачі з незбалансованими класами (95% прикладів одного класу), яка стратегія НЕ допоможе покращити результати?

1) Використання зважених критеріїв розщеплення

2) Збільшення максимальної глибини дерева

3) Застосування техніки SMOTE для балансування даних

4) Модифікація порогу класифікації

2. Чому алгоритм ID3 не підходить для роботи з числовими ознаками без попередньої дискретизації?

1) ID3 не може обчислити ентропію для неперервних значень

2) Числові ознаки завжди мають нульову ентропію

3) ID3 розглядає кожне унікальне значення як окрему категорію

4) Алгоритм не підтримує операції порівняння

3. Яка складність побудови збалансованого дерева рішень для n прикладів з m ознаками?

1) O(n log n)

2) O(mn log n)

3) $O(mn^2)$

4) $O(m^2n)$

4. При реалізації обробки пропущених значень методом суррогатних розщеплень, що є суррогатним розщепленням?

1) Розщеплення за середнім значенням ознаки

2) Альтернативне розщеплення, що найкраще апроксимує основне

3) Випадкове розщеплення прикладів з пропущеними значеннями

4) Розщеплення за найчастішим значенням

5. Чому критерій приросту інформації має зміщення в бік ознак з великою кількістю значень?

1) Більше значень завжди означає більшу інформативність

2) Розщеплення на більше підмножин може випадково зменшити ентропію

3) Алгоритм не може правильно обчислити ентропію для багатьох значень

4) Це помилка в формулі приросту інформації

6. Який метод НЕ використовується для покращення стабільності дерев рішень?

1) Bagging (Bootstrap Aggregating)

2) Random Forest

3) Gradient Boosting

4) Збільшення глибини окремого дерева

7. При використанні дерева рішень для регресії, що зазвичай зберігається в листкових вузлах?

1) Найчастіше значення

2) Середнє значення цільової змінної

3) Медіана значень

4) Мода розподілу

8. Коефіцієнт Gain Ratio був запропонований для вирішення проблеми зміщення в ID3. Як він обчислюється?

1) Приріст інформації / Ентропія ознаки

2) Приріст інформації / Інформація розщеплення

3) Ентропія / Приріст інформації

4) Індекс Джині / Приріст інформації

9. При реалізації CART (Classification and Regression Trees), яка основна відмінність від ID3?

1) CART будує тільки бінарні дерева

2) CART не може працювати з категоріальними даними

3) CART використовує тільки індекс Джині

4) CART не підтримує обрізання

10. Метод Reduced Error Pruning вимагає окремого валідаційного набору даних. Чому?

1) Для обчислення початкової точності

2) Для оцінки покращення точності після видалення піддерев

3) Для навчання додаткових параметрів

4) Для балансування класів

Відповіді:

1. 2 - Збільшення глибини дерева погіршить перенавчання на мажоритарному класі

2. 3 - ID3 трактує кожне унікальне числове значення як окрему категорію, що призводить до надмірного розщеплення

3. 2 - Для кожного з log n рівнів потрібно перевірити m ознак для n прикладів

4. 2 - Суррогатне розщеплення вибирається як найкраще наближення до основного розщеплення

5. 2 - Більша кількість розщеплень підвищує ймовірність випадкового зменшення ентропії

6. 4 - Збільшення глибини окремого дерева погіршує його стабільність

7. 2 - Для регресії в листках зберігається середнє значення цільової змінної

8. 2 - Gain Ratio нормалізує приріст інформації на інформацію розщеплення

9. 1 - CART створює лише бінарні розщеплення, що спрощує структуру дерева

10. 2 - Валідаційний набір потрібен для неупередженої оцінки зміни точності при обрізанні

Завдання для самостійної роботи

1. Реалізуйте модифікацію алгоритму ID3, яка використовує коефіцієнт Gain Ratio замість простого приросту інформації. Порівняйте результати на наборі даних з ознаками, що мають різну кількість можливих значень.

2. Створіть візуалізацію процесу побудови дерева рішень покроково, показуючи, як змінюється структура дерева з додаванням кожного нового розщеплення.

3. Реалізуйте алгоритм C4.5, який підтримує числові ознаки без попередньої дискретизації. Включіть автоматичний пошук оптимальних порогів для розщеплення.

4. Розробіть метод автоматичного визначення оптимальних параметрів обрізання дерева з використанням крос-валідації.

5. Імплементуйте підтримку багатокласової класифікації з використанням стратегії “один проти всіх” та порівняйте з прямою багатокласовою реалізацією.

6. Створіть ensemble метод, який комбінує прогнози кількох дерев рішень, навчених на різних підмножинах ознак.

7. Реалізуйте метод оцінки важливості ознак на основі зменшення домішок (impurity decrease) та візуалізуйте результати.

8. Розробіть адаптивний алгоритм, який автоматично вибирає між різними критеріями розщеплення залежно від характеристик даних.

9. Імплементуйте метод обробки категоріальних ознак з великою кількістю значень через групування рідкісних категорій.

10. Створіть інтерактивний інструмент для дослідження впливу різних гіперпараметрів на структуру та продуктивність дерева рішень.

Теми лабораторних робіт

1. Розробка базової реалізації алгоритму ID3 з підтримкою категоріальних ознак та візуалізацією дерева.

2. Імплементація алгоритму C4.5 з обробкою числових ознак та пропущених значень.

3. Створення системи порівняння різних критеріїв розщеплення (ентропія, Джині, хі-квадрат) на реальних даних.

4. Реалізація методів обрізання дерев (Reduced Error Pruning, Cost Complexity Pruning) з автоматичним вибором параметрів.

5. Розробка алгоритму CART для задач класифікації та регресії з підтримкою суррогатних розщеплень.

6. Створення ансамблевих методів на основі дерев рішень (Random Forest, ExtraTrees).

7. Імплементація інкрементального навчання дерев рішень для потокових даних.

8. Розробка системи інтерпретації дерев рішень з генерацією правил та оцінкою їх надійності.

9. Створення паралельної реалізації побудови дерева рішень для великих наборів даних.

10. Реалізація fuzzy decision trees для роботи з нечіткими даними та невизначеністю.

Класифікація: Random Forest

Теоретичні відомості

Random Forest є потужним ансамблевим методом машинного навчання, який використовує множину дерев рішень для вирішення задач класифікації та регресії. Алгоритм був розроблений Лео Брейманом у 2001 році та швидко став одним з найпопулярніших методів завдяки своїй високій точності, стійкості до перенавчання та здатності працювати з великими обсягами даних.

Основна ідея Random Forest полягає в побудові великої кількості дерев рішень, кожне з яких навчається на випадковій підвибірці даних з використанням випадкової підмножини ознак. Такий підхід, що поєднує bagging (bootstrap aggregating) та випадковий вибір ознак, дозволяє створити різноманітний ансамбль моделей, які разом дають більш точні та стабільні прогнози, ніж окремі дерева.

Bootstrap вибірка - це метод створення нових навчальних наборів шляхом випадкового вибору прикладів з вихідного набору даних з поверненням. Якщо вихідний набір містить n прикладів, то кожна bootstrap вибірка також міститиме n прикладів, але деякі приклади можуть повторюватися, а деякі можуть бути відсутні. Математично ймовірність того, що конкретний приклад не потрапить до bootstrap вибірки, наближається до:

$$P(\text{приклад не вибраний}) = \left(1 - \frac{1}{n}\right)^n \approx \frac{1}{e} \approx 0.368$$

Це означає, що приблизно 63.2% унікальних прикладів потрапляють до кожної bootstrap вибірки, а решта 36.8% формують так звану out-of-bag (OOB) вибірку, яка може використовуватися для оцінки якості моделі без необхідності окремого тестового набору.

Випадковий вибір ознак - на кожному вузлі дерева замість розгляду всіх m доступних ознак для вибору найкращого розщеплення, алгоритм випадково вибирає підмножину з m_{try} ознак. Типові значення:

- Для класифікації: $m_{try} = \lfloor\sqrt{m}\rfloor$
- Для регресії: $m_{try} = \lfloor m/3 \rfloor$

Цей механізм забезпечує додаткову різноманітність дерев та зменшує кореляцію між ними.

Нехай маємо навчальний набір $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, де $x_i \in \mathbb{R}^m$ - вектор ознак, а y_i - мітка класу. Random Forest будує T дерев $\{h_1, h_2, \ldots, h_T\}$, де кожне дерево h_t навчається на bootstrap вибірці D_t.

Для задачі класифікації фінальний прогноз визначається голосуванням:

$$\hat{y} = \text{mode}\{h_1(x), h_2(x), \ldots, h_T(x)\}$$

Для задачі регресії використовується усереднення:

$$\hat{y} = \frac{1}{T}\sum_{t=1}^{T} h_t(x)$$

Оцінка важливості ознак

Random Forest надає два основні методи оцінки важливості ознак:

1. Середнє зменшення домішок (MDI). Для кожної ознаки j обчислюється сума зважених зменшень домішок по всіх вузлах всіх дерев, де ця ознака використовується для розщеплення:

$$\text{MDI}_j = \frac{1}{T}\sum_{t=1}^{T}\sum_{v \in V_t^j} p(v)\Delta_v$$

де V_t^j - множина вузлів в дереві t, що використовують ознаку j, $p(v)$ - частка прикладів, що досягають вузла v, Δ_v - зменшення домішок у вузлі v.

2. Середнє зменшення точності (MDA). Для кожної ознаки вимірюється зменшення точності моделі при випадковій перестановці значень цієї ознаки в OOB прикладах:

$$\mathrm{MDA}_j = \frac{1}{T}\sum_{t=1}^{T}(\mathrm{acc}_t - \mathrm{acc}_{t,j}^{\mathrm{perm}})$$

де acc_t - точність дерева t на OOB прикладах, $\mathrm{acc}_{t,j}^{\mathrm{perm}}$ - точність після перестановки ознаки j.

Розглянемо покроково алгоритм побудови Random Forest для задачі класифікації:

1. Ініціалізація параметрів:

- Кількість дерев T

- Кількість ознак для вибору на кожному розщепленні m_{try}

- Максимальна глибина дерев d_{max}

- Мінімальна кількість прикладів для розщеплення n_{min}

2. Побудова ансамблю дерев: Для кожного дерева t від 1 до T:

- Створити bootstrap вибірку D_t розміру n з вихідного набору D

- Побудувати дерево рішень h_t на вибірці D_t з модифікованим алгоритмом:

- На кожному вузлі випадково вибрати m_{try} ознак з m доступних

- Знайти найкраще розщеплення серед вибраних ознак

- Продовжувати рекурсивно до досягнення критеріїв зупинки

3. Агрегація прогнозів: Для нового прикладу x:

- Отримати прогноз від кожного дерева: $h_t(x)$

- Визначити фінальний прогноз голосуванням

Псевдокод алгоритму Random Forest

```
функція НавчитиRandomForest(D, T, m_try, d_max, n_min)
    // D - навчальний набір даних
    // T - кількість дерев
    // m_try - кількість ознак для вибору
    // d_max - максимальна глибина
    // n_min - мінімальна кількість прикладів

    ліс = []
    oob_приклади = [] // для кожного дерева
```

```
    для t = 1 до T виконати
        // Створення bootstrap вибірки
        D_t, OOB_t = СтворитиBootstrapВибірку(D)

        // Побудова дерева з випадковим вибором ознак
        дерево = ПобудуватиВипадковеДерево(D_t, m_try, d_max, n_min)

        ліс.додати(дерево)
        oob_приклади.додати(OOB_t)

    повернути ліс, oob_приклади

функція СтворитиBootstrapВибірку(D)
    n = |D|
    D_bootstrap = []
    використані = новий масив[n] з false

    // Вибір n прикладів з поверненням
    для i = 1 до n виконати
        індекс = випадкове_число(0, n-1)
        D_bootstrap.додати(D[індекс])
        використані[індекс] = true

    // Формування OOB вибірки
    OOB = []
    для i = 0 до n-1 виконати
        якщо не використані[i] тоді
            OOB.додати(D[i])

    повернути D_bootstrap, OOB

функція ПобудуватиВипадковеДерево(D, m_try, d_max, n_min, глибина = 0
)
    // Критерії зупинки
    якщо |D| < n_min або глибина >= d_max тоді
        повернути СтворитиЛисток(D)

    якщо всі приклади одного класу тоді
        повернути СтворитиЛисток(D)

    // Випадковий вибір ознак
    всі_ознаки = отримати_всі_ознаки(D)
    вибрані_ознаки = випадкова_вибірка(всі_ознаки, m_try)
```

```
    // Пошук найкращого розщеплення серед вибраних ознак
    найкраща_ознака = null
    найкращий_поріг = null
    найкращий_приріст = -∞

    для кожної ознаки з вибрані_ознаки виконати
        поріг, приріст = ЗнайтиНайкращийПоріг(D, ознака)
        якщо приріст > найкращий_приріст тоді
            найкраща_ознака = ознака
            найкращий_поріг = поріг
            найкращий_приріст = приріст

    // Розщеплення даних
    D_ліво, D_право = РозщепитиДані(D, найкраща_ознака, найкращий_пор
іг)

    // Рекурсивна побудова піддерев
    вузол = новий ВнутрішнійВузол()
    вузол.ознака = найкраща_ознака
    вузол.поріг = найкращий_поріг
    вузол.ліве = ПобудуватиВипадковеДерево(D_ліво, m_try, d_max, n_mi
n, глибина + 1)
    вузол.праве = ПобудуватиВипадковеДерево(D_право, m_try, d_max, n_
min, глибина + 1)

    повернути вузол

функція КласифікуватиRandomForest(ліс, x)
    голоси = новий словник()

    для кожного дерева з ліс виконати
        прогноз = КласифікуватиДерево(дерево, x)
        якщо прогноз в голоси тоді
            голоси[прогноз] += 1
        інакше
            голоси[прогноз] = 1

    // Повернути клас з максимальною кількістю голосів
    повернути argmax(голоси)

функція ОцінитиВажливістьОзнак(ліс, oob_приклади, m)
    важливість = новий масив[m] з 0

    для t = 1 до |ліс| виконати
        дерево = ліс[t-1]
```

```
        OOB_t = oob_приклади[t-1]

        // Базова точність на OOB
        точність_база = ОцінитиТочність(дерево, OOB_t)

        для j = 0 до m-1 виконати
            // Перемішати значення ознаки j
            OOB_perm = ПеремішатиОзнаку(OOB_t, j)
            точність_perm = ОцінитиТочність(дерево, OOB_perm)

            // Накопичити зменшення точності
            важливість[j] += (точність_база - точність_perm)

    // Нормалізувати
    для j = 0 до m-1 виконати
        важливість[j] /= |ліс|

    повернути важливість
```

Приклад застосування

Розглянемо детальний приклад побудови Random Forest для задачі класифікації типу ірису на основі характеристик квітки. Маємо навчальний набір з 12 прикладів, 4 ознаками та 3 класами.

Навчальні дані:

- Ознаки: довжина пелюстки (ДП), ширина пелюстки (ШП), довжина чашолистка (ДЧ), ширина чашолистка (ШЧ)

- Класи: Setosa (S), Versicolor (Ve), Virginica (Vi)

Приклади (спрощені значення):

1. (5.1, 3.5, 1.4, 0.2) → S

2. (4.9, 3.0, 1.4, 0.2) → S

3. (7.0, 3.2, 4.7, 1.4) → Ve

4. (6.4, 3.2, 4.5, 1.5) → Ve

5. (6.3, 3.3, 6.0, 2.5) → Vi

6. (5.8, 2.7, 5.1, 1.9) → Vi

7. (4.6, 3.1, 1.5, 0.2) → S

8. (5.0, 3.6, 1.4, 0.2) → S

9. (6.5, 2.8, 4.6, 1.5) → Ve

10. (5.7, 2.8, 4.5, 1.3) → Ve

11. (7.1, 3.0, 5.9, 2.1) → Vi

12. (6.7, 3.0, 5.2, 2.3) → Vi

Параметри Random Forest:

- Кількість дерев: T = 3
- Кількість ознак для вибору: m_try = 2 (з 4 доступних)
- Максимальна глибина: d_max = 3

Побудова першого дерева:

Крок 1.1: Bootstrap вибірка Випадково вибираємо 12 прикладів з поверненням: Bootstrap_1: [1, 1, 3, 4, 5, 7, 8, 9, 9, 10, 11, 12] OOB_1: [2, 6] (приклади, що не потрапили до вибірки)

Крок 1.2: Побудова кореневого вузла Випадково вибираємо 2 ознаки з 4: ДП, ШЧ

Для ознаки ДП знаходимо оптимальний поріг:

- Сортуємо значення: [1.4, 1.4, 1.5, 4.5, 4.6, 4.6, 4.7, 5.2, 5.9, 6.0]
- Найкращий поріг: 3.0 (розділяє Setosa від інших)

Для ознаки ШЧ:

- Сортуємо значення: [0.2, 0.2, 0.2, 1.3, 1.4, 1.5, 1.5, 2.1, 2.3, 2.5]
- Найкращий поріг: 0.8

Приріст інформації для ДП з порогом 3.0 вищий, тому вибираємо це розщеплення.

Крок 1.3: Рекурсивна побудова

Ліва гілка (ДП ⩽ 3.0): всі приклади класу S → листок “S”

Права гілка (ДП > 3.0): суміш Ve та Vi, продовжуємо побудову…

Побудова другого дерева:

Bootstrap_2: [2, 3, 3, 5, 6, 7, 8, 10, 11, 11, 12, 12] OOB_2: [1, 4, 9]

Аналогічно будуємо друге дерево з іншою bootstrap вибіркою та випадковим вибором ознак.

Побудова третього дерева:

Bootstrap_3: [1, 2, 4, 5, 6, 6, 7, 9, 10, 11, 12, 12] OOB_3: [3, 8]

Класифікація нового прикладу:

Нехай потрібно класифікувати новий приклад: (6.0, 3.0, 4.8, 1.6)

Дерево 1: проходимо шлях згідно з побудованою структурою → прогноз: Ve Дерево 2: → прогноз: Ve Дерево 3: → прогноз: Vi

Фінальний прогноз голосуванням: Ve (2 голоси проти 1)

Оцінка важливості ознак:

Використовуючи OOB приклади для кожного дерева:

- Для дерева 1 з OOB_1 = [2, 6]

- Базова точність: 100% (правильно класифікує обидва)

- Перемішуємо ДП: точність падає до 50%

- Зменшення точності для ДП: 0.5

Аналогічно для всіх ознак та дерев, отримуємо середню важливість:

- ДП: 0.45 (найважливіша)

- ШЧ: 0.30

- ДЧ: 0.15

- ШП: 0.10

Область застосування

Random Forest успішно застосовується в широкому спектрі задач завдяки своїй універсальності та високій точності:

Біоінформатика та медицина - алгоритм використовується для аналізу генетичних даних, прогнозування взаємодії білків, класифікації типів раку на основі експресії генів. Здатність оцінювати важливість ознак особливо цінна при роботі з високорозмірними біологічними даними.

Фінансова аналітика - Random Forest застосовується для оцінки кредитних ризиків, виявлення шахрайських транзакцій, прогнозування банкрутства компаній. Стійкість до викидів та здатність моделювати нелінійні залежності роблять його ідеальним для фінансових даних.

Комп’ютерний зір - алгоритм використовується для розпізнавання об’єктів, сегментації зображень, виявлення аномалій на медичних знімках. Паралельна природа Random Forest дозволяє ефективно обробляти великі обсяги візуальних даних.

Екологічне моделювання - прогнозування поширення видів, оцінка якості води та повітря, моделювання кліматичних змін. Здатність працювати з

різнотипними даними та враховувати складні взаємодії між факторами робить Random Forest популярним у екологічних дослідженнях.

Рекомендаційні системи - персоналізація контенту, прогнозування вподобань користувачів, ранжування результатів пошуку. Ансамблевий підхід допомагає створювати стабільні рекомендації навіть при розрідженості даних.

Промислова діагностика - прогнозування відмов обладнання, оптимізація технологічних процесів, контроль якості продукції. Random Forest ефективно працює з даними сенсорів та історичними записами про технічне обслуговування.

Приклади застосування

Приклад 1: Діагностика діабету

Медичний дослідницький центр використовує Random Forest для раннього виявлення діабету 2 типу. Модель аналізує 20 різних показників: - Біохімічні: рівень глюкози натще, HbA1c, інсулін - Фізіологічні: індекс маси тіла, артеріальний тиск, вік - Генетичні маркери та сімейна історія

Random Forest з 500 деревами досягає точності 92% на тестовій вибірці. Аналіз важливості ознак показує, що найбільш інформативними є рівень HbA1c (28%), індекс маси тіла (19%) та вік (15%). Це дозволяє лікарям зосередитися на ключових факторах ризику при скринінгу пацієнтів.

Приклад 2: Прогнозування відтоку клієнтів в e-commerce

Інтернет-магазин впроваджує Random Forest для прогнозування ймовірності того, що клієнт припинить робити покупки. Модель враховує: - Історію покупок: частота, середній чек, категорії товарів - Поведінку на сайті: час сесій, перегляди сторінок, використання пошуку - Взаємодію з маркетингом: відкриття email, участь в акціях - Демографічні дані та географію

Система з 300 деревами ідентифікує 78% клієнтів, які відмовляться від покупок протягом наступних 3 місяців. На основі прогнозів маркетингова команда розробляє персоналізовані програми утримання, що знижує відтік на 23%.

Приклад 3: Класифікація дефектів у виробництві мікросхем

Завод з виробництва напівпровідників використовує Random Forest для автоматичної класифікації дефектів на основі зображень з електронного мікроскопа. Система аналізує: - Текстурні характеристики: ентропія,

контраст, однорідність - Геометричні параметри: площа, периметр, округлість дефекту - Контекстну інформацію: розташування на пластині, етап виробництва

Random Forest з 200 деревами класифікує дефекти на 12 категорій з точністю 95%, що дозволяє швидко виявляти проблеми в технологічному процесі та зменшити відсоток браку на 30%.

Переваги та недоліки

Переваги

Висока точність прогнозування - Random Forest часто демонструє одну з найкращих продуктивностей серед алгоритмів машинного навчання без необхідності тонкого налаштування. Ансамблевий підхід ефективно зменшує варіацію окремих дерев, забезпечуючи стабільні результати.

Стійкість до перенавчання - використання bootstrap вибірок та випадкового вибору ознак створює декорельовані дерева, що значно знижує ризик перенавчання порівняно з окремим глибоким деревом рішень.

Робота з різнотипними даними - алгоритм природно обробляє як числові, так і категоріальні ознаки без необхідності попередньої обробки. Немає потреби в нормалізації або стандартизації даних.

Ефективна обробка пропущених значень - Random Forest може використовувати різні стратегії для роботи з пропущеними даними, включаючи proximity-based imputation та surrogate splits.

Оцінка важливості ознак - вбудовані механізми оцінки важливості надають цінну інформацію про те, які ознаки найбільше впливають на прогнози, що допомагає в інтерпретації моделі та відборі ознак.

Паралелізація - побудова окремих дерев може виконуватися незалежно, що дозволяє ефективно використовувати багатоядерні процесори та розподілені обчислювальні системи.

Out-of-bag оцінка - OOB помилка надає неупереджену оцінку узагальнюючої здатності моделі без необхідності окремого валідаційного набору.

Стійкість до викидів та шуму - ансамблевий підхід та голосування роблять модель менш чутливою до аномальних значень у даних.

Недоліки

Великий розмір моделі - збереження сотень або тисяч дерев вимагає значної пам’яті, що може бути проблемою для вбудованих систем або мобільних пристроїв.

Повільне прогнозування для великих ансамблів - необхідність обходу всіх дерев для кожного прогнозу може бути повільною при великій кількості дерев, особливо для real-time застосувань.

Складність інтерпретації окремих прогнозів - хоча важливість ознак зрозуміла на глобальному рівні, пояснити конкретне рішення для окремого прикладу складно через агрегацію багатьох дерев.

Зміщення в бік мажоритарного класу - при сильно незбалансованих даних Random Forest може недостатньо добре класифікувати рідкісні класи без додаткових технік балансування.

Екстраполяція за межі навчальних даних - як і окремі дерева рішень, Random Forest не може робити прогнози за межами діапазону значень у навчальних даних, що обмежує його використання для трендового аналізу.

Чутливість до кореляції ознак - при наявності багатьох сильно корельованих ознак алгоритм може недооцінювати їх індивідуальну важливість.

Складність налаштування гіперпараметрів - хоча базова конфігурація часто працює добре, оптимізація параметрів для конкретної задачі може вимагати значних обчислювальних ресурсів.

Контрольні запитання

1. Поясніть, як bootstrap вибірка сприяє зменшенню перенавчання в Random Forest.

2. Чому випадковий вибір підмножини ознак на кожному вузлі важливий для ефективності Random Forest?

3. Як обчислюється out-of-bag помилка та чому вона є неупередженою оцінкою узагальнюючої здатності?

4. Опишіть різницю між Mean Decrease in Impurity та Mean Decrease in Accuracy для оцінки важливості ознак.

5. Як Random Forest обробляє категоріальні ознаки з великою кількістю рівнів?

6. Чому Random Forest менш схильний до перенавчання порівняно з окремим глибоким деревом рішень?

7. Як вибір кількості ознак m_try впливає на баланс між зміщенням та варіацією моделі?

8. Поясніть, чому Random Forest може бути неефективним для даних з великою кількістю нерелевантних ознак.

9. Як можна адаптувати Random Forest для роботи з незбалансованими класами?

10. Які модифікації Random Forest існують для покращення швидкості прогнозування?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Random Forest для задач класифікації та регресії

2. Мета

Освоїти принципи побудови ансамблевих моделей, навчитися реалізовувати Random Forest з нуля, зрозуміти механізми bootstrap вибірки та випадкового вибору ознак.

3. Завдання

- Реалізувати генерацію bootstrap вибірок

- Створити модифіковане дерево рішень з випадковим вибором ознак

- Реалізувати ансамбль дерев з механізмами голосування та усереднення

- Додати обчислення OOB помилки

- Реалізувати оцінку важливості ознак

- Провести експерименти з різними конфігураціями

4. Опис

Студенти розробляють повну реалізацію Random Forest, включаючи всі ключові компоненти: bootstrap sampling, модифіковані дерева рішень, агрегацію прогнозів, OOB оцінку та аналіз важливості ознак. Реалізація повинна підтримувати як задачі класифікації, так і регресії, ефективно працювати з різними типами даних та надавати можливості для аналізу моделі.

5. Порядок виконання

1. Реалізувати функцію генерації bootstrap вибірок з відстеженням OOB прикладів

2. Модифікувати алгоритм побудови дерева рішень для випадкового вибору m_try ознак

3. Створити клас RandomForest з методами fit та predict

4. Реалізувати паралельну побудову дерев для прискорення навчання

5. Додати методи обчислення OOB score та confusion matrix

6. Реалізувати два методи оцінки важливості ознак (MDI та MDA)

7. Створити візуалізацію результатів та важливості ознак

8. Провести порівняння з одиночним деревом рішень

6. Результат

Функціональна реалізація Random Forest з можливістю навчання на різних типах даних, ефективним прогнозуванням, оцінкою якості через OOB та детальним аналізом важливості ознак.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функція генерації bootstrap вибірок

- Клас модифікованого дерева рішень з випадковим вибором ознак

- Клас RandomForest з методами навчання та прогнозування

- Функції обчислення OOB помилки

- Методи оцінки важливості ознак (MDI та MDA)

- Паралельна реалізація побудови дерев

2. Продемонструвати роботу додатку:

- Навчання Random Forest на різних наборах даних

- Порівняння точності з різною кількістю дерев

- Візуалізація OOB помилки залежно від кількості дерев

- Аналіз важливості ознак для реальних даних

- Порівняння часу навчання послідовної та паралельної версій

3. Бути готовим відповісти на питання:

- Як змінюється OOB помилка зі збільшенням кількості дерев?

- Який вплив має параметр m_try на якість моделі?

- Як реалізована паралелізація процесу навчання?

- Чому важливість ознак може відрізнятися між MDI та MDA методами?

Тести

1. Типове значення параметра m_try для задачі класифікації з 16 ознаками:

1) 16

2) 8

3) 4

4) 1

2. Ймовірність того, що конкретний приклад НЕ потрапить до bootstrap вибірки наближається до:

1) 0.5

2) 0.368

3) 0.632

4) 0.25

3. OOB (Out-of-Bag) приклади використовуються для:

1) Навчання додаткових дерев

2) Неупередженої оцінки помилки

3) Збільшення розміру навчальної вибірки

4) Прискорення навчання

4. Основна причина декореляції дерев у Random Forest:

1) Використання різних алгоритмів побудови

2) Випадковий вибір підмножини ознак

3) Обмеження глибини дерев

4) Нормалізація даних

5. При регресії Random Forest агрегує прогнози дерев шляхом:

1) Вибору медіани

2) Голосування

3) Усереднення

4) Вибору моди

Відповіді:

1-3,

2-2,

3-2,

4-2,

5-3

Тести підвищеної складності

1. У Random Forest з 1000 деревами, якщо перші 100 дерев вже забезпечують високу точність, чому варто продовжувати додавати дерева?

1) Для зменшення часу прогнозування

2) Для покращення OOB оцінки та стабільності важливості ознак

3) Для збільшення швидкості навчання

4) Для зменшення використання пам’яті

2. При використанні Random Forest для незбалансованих даних (1:100), яка модифікація НЕ допоможе?

1) Balanced Random Forest з різними bootstrap розмірами для класів

2) Зважене голосування на основі частот класів

3) Збільшення кількості дерев до 10000

4) Stratified bootstrap sampling

3. Чому Mean Decrease in Accuracy часто вважається більш надійною мірою важливості ніж Mean Decrease in Impurity?

1) MDI завищує важливість ознак з великою кількістю рівнів

2) MDA працює швидше

3) MDI не враховує OOB приклади

4) MDA використовує всі дерева

4. Feature bagging в Random Forest найбільш ефективний коли:

1) Всі ознаки незалежні

2) Є групи сильно корельованих ознак

3) Кількість ознак мала

4) Дані лінійно роздільні

5. Extremely Randomized Trees відрізняються від Random Forest тим, що:

1) Використовують більше дерев

2) Вибирають пороги розщеплення випадково

3) Не використовують bootstrap

4) Обмежують глибину дерев

6. При реалізації Random Forest для big data, який підхід НЕ є ефективним?

1) Розподілена побудова дерев на кластері

2) Використання підвибірки для кожного дерева

3) Збереження всіх дерев в оперативній пам’яті

4) Квантування числових ознак

7. Proximities в Random Forest обчислюються як:

1) Евклідова відстань між прикладами

2) Частка дерев, де приклади потрапляють в один листок

3) Кореляція між прогнозами

4) Косинусна подібність векторів ознак

8. Для часових рядів Random Forest модифікується використанням:

1) Спеціальних функцій втрат

2) Block bootstrap замість звичайного

3) Рекурентних дерев

4) Градієнтного спуску

9. Variable hunting з Random Forest найефективніший при:

1) Малій кількості справді важливих ознак серед тисяч

2) Всіх ознаках однакової важливості

3) Лінійних залежностях

4) Малій кількості прикладів

10. Conditional Random Forest вирішує проблему:

1) Повільного прогнозування

2) Зміщення відбору змінних при корельованих ознаках

3) Великого розміру моделі

4) Неможливості екстраполяції

Відповіді:

1. 2 - Додаткові дерева покращують стабільність OOB оцінок та оцінок важливості ознак

2. 3 - Просте збільшення кількості дерев не вирішує проблему дисбалансу класів

3. 1 - MDI має тенденцію завищувати важливість категоріальних ознак з багатьма рівнями

4. 2 - При корельованих ознаках випадковий вибір допомагає використати різні групи ознак

5. 2 - Extra Trees вибирають пороги випадково, а не оптимально

6. 3 - Для big data слід використовувати ефективні структури даних, а не зберігати все в RAM

7. 2 - Proximity визначається частотою потрапляння прикладів в однакові листки

8. 2 - Block bootstrap зберігає часову структуру даних

9. 1 - Random Forest ефективно знаходить важливі ознаки серед великої кількості нерелевантних

10. 2 - Conditional inference trees усувають зміщення при виборі корельованих предикторів

Завдання для самостійної роботи

1. Реалізуйте варіант Random Forest з адаптивним вибором m_try для кожного дерева на основі OOB помилки попередніх дерев.

2. Створіть модифікацію Balanced Random Forest для роботи з сильно незбалансованими даними, використовуючи різні стратегії вибірки для різних класів.

3. Реалізуйте метод виявлення взаємодій між ознаками на основі аналізу структури дерев у Random Forest.

4. Розробіть алгоритм стиснення Random Forest шляхом видалення найменш важливих дерев без значної втрати точності.

5. Імплементуйте Quantile Random Forest для отримання прогнозних інтервалів замість точкових оцінок.

6. Створіть візуалізацію “шляхів прийняття рішень” для конкретного прикладу через всі дерева ансамблю.

7. Реалізуйте online версію Random Forest, яка може донавчатися на нових даних без повної перебудови.

8. Розробіть метод локальної інтерпретації прогнозів Random Forest на основі LIME або SHAP підходів.

9. Імплементуйте GPU-прискорену версію Random Forest з використанням CUDA або OpenCL.

10. Створіть гібридну модель, що комбінує Random Forest з градієнтним бустингом для покращення точності.

Теми лабораторних робіт

1. Базова реалізація Random Forest з підтримкою класифікації та регресії, включаючи OOB оцінку.

2. Розробка системи автоматичного підбору гіперпараметрів Random Forest з використанням байєсівської оптимізації.

3. Імплементація розподіленого Random Forest для обробки великих даних з використанням Apache Spark.

4. Створення інтерактивної системи аналізу важливості ознак з можливістю ітеративного відбору.

5. Реалізація Proximity-based методів для виявлення аномалій та кластеризації з Random Forest.

6. Розробка ансамблю Random Forest моделей для multi-label класифікації.

7. Створення системи пояснення прогнозів Random Forest з генерацією правил та контрфактичних прикладів.

8. Імплементація варіантів Random Forest для обробки потокових даних з concept drift.

9. Розробка методів візуалізації високорозмірних даних на основі Random Forest proximities.

10. Створення фреймворку для порівняння різних варіантів Random Forest (Extra Trees, Conditional RF, Quantile RF).

Класифікація: Gradient Boosting

Теоретичні відомості

Gradient Boosting є потужним ансамблевим методом машинного навчання, який будує прогнозну модель шляхом послідовного додавання слабких

учнів, кожен з яких намагається виправити помилки попередніх. На відміну від Random Forest, де дерева будуються незалежно та паралельно, Gradient Boosting створює дерева послідовно, причому кожне наступне дерево навчається на залишках попередніх прогнозів.

Алгоритм був формалізований Джеромом Фрідманом у 1999-2001 роках та швидко став одним з найефективніших методів для широкого спектру задач машинного навчання. Основна ідея полягає в оптимізації довільної диференційовної функції втрат шляхом ітеративного додавання моделей, які апроксимують градієнт цієї функції.

Нехай маємо навчальний набір даних $\{(x_i, y_i)\}_{i=1}^n$, де $x_i \in \mathbb{R}^d$ - вектор ознак, y_i - цільова змінна. Мета полягає в знаходженні функції $F(x)$, яка мінімізує очікуване значення функції втрат $L(y, F(x))$:

$$F^* = \arg\min_F \mathbb{E}_{x,y}[L(y, F(x))]$$

Gradient Boosting апроксимує F^* адитивною моделлю виду:

$$F_M(x) = \sum_{m=0}^{M} f_m(x)$$

де $f_0(x)$ - початкове наближення, а $f_m(x)$ для $m \geq 1$ - слабкі учні (зазвичай дерева рішень).

Градієнтний спуск у функціональному просторі

Ключова ідея Gradient Boosting полягає в інтерпретації процесу побудови моделі як градієнтного спуску у функціональному просторі. На кожній ітерації m ми хочемо знайти функцію f_m, яка найкраще наближає антиградієнт функції втрат:

$$f_m = \arg\min_f \sum_{i=1}^{n} (-g_{im} - f(x_i))^2$$

де g_{im} - градієнт функції втрат для i-го прикладу на ітерації m:

$$g_{im} = \left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}}$$

Цей градієнт показує напрямок, в якому потрібно змінити прогноз для кожного прикладу, щоб максимально зменшити функцію втрат.

Функції втрат

Різні функції втрат призводять до різних варіантів Gradient Boosting:

Для регресії:

- Квадратична втрата: $L(y, F) = \frac{1}{2}(y - F)^2$

Градієнт: $g = -(y - F) = F - y$ (залишок)

- Абсолютна втрата: $L(y, F) = |y - F|$

Градієнт: $g = \text{sign}(F - y)$

- Втрата Хубера: $L(y, F) = \begin{cases} \frac{1}{2}(y - F)^2 & \text{якщо } |y - F| \leq \delta \\ \delta|y - F| - \frac{1}{2}\delta^2 & \text{інакше} \end{cases}$

Для бінарної класифікації:

- Логістична втрата: $L(y, F) = \log(1 + \exp(-2yF))$, де $y \in \{-1,1\}$

Градієнт: $g = \frac{-2y}{1+\exp(2yF)}$

- Експоненційна втрата: $L(y, F) = \exp(-yF)$

Градієнт: $g = -y\exp(-yF)$

Регуляризація

Для запобігання перенавчанню Gradient Boosting використовує кілька технік регуляризації:

Shrinkage (навчальна швидкість) - кожне дерево додається з коефіцієнтом $\nu \in (0,1]$:

$$F_m(x) = F_{m-1}(x) + \nu \cdot f_m(x)$$

Менші значення ν зазвичай призводять до кращої узагальнюючої здатності, але вимагають більше ітерацій.

Subsampling - на кожній ітерації використовується випадкова підвибірка розміру $fraction \cdot n$ з навчальних даних. Це створює стохастичний градієнтний бустинг, який часто покращує результати.

Обмеження складності дерев - використання неглибоких дерев (глибина 3-8) як базових учнів. Дерева з одним розщепленням (stumps) відповідають адитивній моделі без взаємодій між ознаками.

Розглянемо покроковий алгоритм для загального випадку:

1. Ініціалізація: Знайти початкове наближення $F_0(x)$, яке мінімізує функцію втрат:

$$F_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} L\left(y_i, \gamma\right)$$

2. Ітеративна побудова: Для $m = 1$ до M:

1) Обчислити псевдо-залишки (негативні градієнти):

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F=F_{m-1}} \quad \text{для } i = 1, \ldots, n$$

2) Навчити базовий учень $h_m(x)$ на парах (x_i, r_{im})

3) Знайти оптимальний крок γ_m:

$$\gamma_m = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right)$$

4) Оновити модель:

$$F_m(x) = F_{m-1}(x) + \nu \cdot \gamma_m \cdot h_m(x)$$

3. Фінальна модель: $F_M(x)$

Gradient Boosting для класифікації

Для задач багатокласової класифікації з K класами використовується K окремих моделей, по одній для кожного класу. Ймовірність належності до класу k обчислюється через softmax:

$$p_k(x) = \frac{\exp(F_k(x))}{\sum_{j=1}^{K} \exp\left(F_j(x)\right)}$$

Функція втрат - це негативна log-likelihood:

$$L(\{y_{ik}\}, \{F_k\}) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log p_k(x_i)$$

де $y_{ik} = 1$, якщо приклад i належить до класу k, інакше 0.

Псевдокод алгоритму Gradient Boosting

```
функція GradientBoosting(дані, M, ν, функція_втрат, глибина_дерева)
    // дані - навчальний набір {(x_i, y_i)}
    // M - кількість ітерацій
    // ν - навчальна швидкість
    // функція_втрат - L(y, F)
    // глибина_дерева - максимальна глибина базових учнів

    // Ініціалізація
    F_0 = ІніціалізуватиМодель(дані, функція_втрат)
```

```
    моделі = [F_0]

    для m = 1 до M виконати
        // Обчислити псевдо-залишки
        залишки = []
        для i = 1 до n виконати
            g_i = ОбчислитиГрадієнт(функція_втрат, y_i, F_{m-1}(x_i))
            залишки.додати((x_i, -g_i))

        // Навчити базовий учень
        h_m = НавчитиДерево(залишки, глибина_дерева)

        // Знайти оптимальний крок
        γ_m = ОптимальнийКрок(дані, F_{m-1}, h_m, функція_втрат)

        // Оновити модель
        F_m = функція(x) { повернути F_{m-1}(x) + ν * γ_m * h_m(x) }
        моделі.додати(F_m)

    повернути F_M

функція ОбчислитиГрадієнт(L, y, F)
    якщо L є квадратична_втрата тоді
        повернути F - y
    якщо L є логістична_втрата тоді
        p = 1 / (1 + exp(-F))
        повернути p - y
    якщо L є абсолютна_втрата тоді
        повернути sign(F - y)
    // інші функції втрат...

функція ОптимальнийКрок(дані, F_поточна, h, L)
    // Одновимірна оптимізація
    функція_цілі = функція(γ) {
        сума = 0
        для (x_i, y_i) в дані виконати
            сума += L(y_i, F_поточна(x_i) + γ * h(x_i))
        повернути сума
    }

    повернути ЛінійнийПошук(функція_цілі, 0, 1)

функція НавчитиДерево(дані, макс_глибина)
    повернути ПобудуватиРегресійнеДерево(дані, макс_глибина)
```

```
// Для класифікації
функція GradientBoostingКласифікація(дані, M, ν, K)
    // K - кількість класів
    моделі = []

    для k = 1 до K виконати
        // Створити бінарні мітки для класу k
        дані_k = []
        для (x_i, y_i) в дані виконати
            y_ik = якщо y_i == k тоді 1 інакше 0
            дані_k.додати((x_i, y_ik))

        // Навчити модель для класу k
        F_k = GradientBoosting(дані_k, M, ν, логістична_втрата, 5)
        моделі.додати(F_k)

    повернути моделі

функція Прогнозувати(моделі, x)
    // Для регресії
    якщо моделі.довжина == 1 тоді
        повернути моделі[0](x)

    // Для класифікації
    scores = []
    для F_k в моделі виконати
        scores.додати(F_k(x))

    // Softmax
    exp_scores = [exp(s) для s в scores]
    сума = sum(exp_scores)
    ймовірності = [e / сума для e в exp_scores]

    повернути argmax(ймовірності)
```

Стохастичний Gradient Boosting

```
функція СтохастичнийGradientBoosting(дані, M, ν, subsample_rate)
    F_0 = ІніціалізуватиМодель(дані)

    для m = 1 до M виконати
        // Випадкова підвибірка
        n_subsample = floor(n * subsample_rate)
        індекси = випадкова_вибірка(1..n, n_subsample)
        дані_підвибірка = [дані[i] для i в індекси]
```

```
        // Обчислити градієнти тільки для підвибірки
        залишки = []
        для (x_i, y_i) в дані_підвибірка виконати
            g_i = ОбчислитиГрадієнт(L, y_i, F_{m-1}(x_i))
            залишки.додати((x_i, -g_i))

        // Навчити на підвибірці
        h_m = НавчитиДерево(залишки, глибина)

        // Оновити модель для всіх даних
        F_m = F_{m-1} + ν * h_m

    повернути F_M
```

Приклад застосування

Розглянемо детальний приклад застосування Gradient Boosting для задачі регресії. Маємо простий набір даних для прогнозування ціни будинку на основі його площі.

Навчальні дані:

1. (50 м², 100 тис.)

2. (60 м², 150 тис.)

3. (70 м², 160 тис.)

4. (80 м², 200 тис.)

5. (90 м², 250 тис.)

6. (100 м², 280 тис.)

Використаємо квадратичну функцію втрат та побудуємо 3 ітерації з деревами глибини 1 (stumps).

Ітерація 0: Ініціалізація

Для квадратичної втрати початкове наближення - це середнє значення:

$$F_0(x) = \frac{1}{6}(100 + 150 + 160 + 200 + 250 + 280) = 190$$

Ітерація 1:

Крок 1.1: Обчислення залишків

- $r_1 = 100 - 190 = -90$

- $r_2 = 150 - 190 = -40$

- $r_3 = 160 - 190 = -30$

- $r_4 = 200 - 190 = 10$

- $r_5 = 250 - 190 = 60$

- $r_6 = 280 - 190 = 90$

Крок 1.2: Побудова дерева Найкраще розщеплення для дерева глибини 1: площа ⩽ 75 м²

Ліва гілка (⩽ 75):

середнє = $\frac{-90+(-40)+(-30)}{3} = -53.33$

Права гілка (> 75):

середнє = $\frac{10+60+90}{3} = 53.33$

Крок 1.3: Оновлення моделі З навчальною швидкістю $\nu = 0.1$:

$$F_1(x) = 190 + 0.1 \cdot h_1(x)$$

Де $h_1(x) = \begin{cases} -53.33 & \text{якщо площа} \leq 75 \\ 53.33 & \text{якщо площа} > 75 \end{cases}$

Нові прогнози:

- Для площі ⩽ 75: $190 + 0.1 \cdot (-53.33) = 184.67$

- Для площі > 75: $190 + 0.1 \cdot 53.33 = 195.33$

Ітерація 2:

Крок 2.1: Нові залишки

- $r_1 = 100 - 184.67 = -84.67$

- $r_2 = 150 - 184.67 = -34.67$

- $r_3 = 160 - 184.67 = -24.67$

- $r_4 = 200 - 195.33 = 4.67$

- $r_5 = 250 - 195.33 = 54.67$

- $r_6 = 280 - 195.33 = 84.67$

Крок 2.2: Нове дерево Оптимальне розщеплення: площа ⩽ 55 м²

Ліва гілка:

середнє = -84.67

Права гілка:

середнє = $\frac{-34.67+(-24.67)+4.67+54.67+84.67}{5} = 16.93$

Крок 2.3: Оновлення

$$F_2(x) = F_1(x) + 0.1 \cdot h_2(x)$$

Ітерація 3:

Процес продовжується аналогічно. Кожне нове дерево фокусується на виправленні залишкових помилок попередніх ітерацій.

Фінальна модель:

$$F_3(x) = 190 + 0.1 \cdot h_1(x) + 0.1 \cdot h_2(x) + 0.1 \cdot h_3(x)$$

Для прогнозування ціни нового будинку площею 75 м²:

1. Початкове наближення: 190

2. Корекція від h_1: площа = 75, тому використовуємо ліву гілку: -53.33

3. Корекція від h_2: використовуємо праву гілку: 16.93

4. Корекція від h_3: (залежить від побудованого дерева)

Фінальний прогноз ≈ 190 + 0.1×(-53.33) + 0.1×16.93 + … ≈ 186.36 тис.

Область застосування

Gradient Boosting є одним з найефективніших алгоритмів машинного навчання та широко застосовується в різних галузях:

Змагання з машинного навчання - алгоритм домінує на платформах як Kaggle, часто забезпечуючи найкращі результати. Популярні реалізації XGBoost, LightGBM та CatBoost стали стандартними інструментами для data scientists.

Пошукові системи - використовується для ранжування результатів пошуку. Алгоритм може враховувати сотні ознак для визначення релевантності документів запиту користувача.

Фінансове моделювання - прогнозування цін акцій, оцінка кредитних ризиків, виявлення шахрайства. Здатність моделювати складні нелінійні залежності робить його ідеальним для фінансових даних.

Рекомендаційні системи - персоналізація контенту та товарів. Gradient Boosting ефективно комбінує різні сигнали користувацької поведінки для прогнозування вподобань.

Медична діагностика - прогнозування ризиків захворювань, аналіз медичних зображень, персоналізація лікування. Висока точність критична для медичних застосувань.

Обробка природної мови - класифікація текстів, визначення тональності, машинний переклад. Часто використовується в поєднанні з нейронними мережами.

Комп'ютерний зір - детекція об'єктів, розпізнавання облич, сегментація зображень. Gradient Boosting може працювати з hand-crafted ознаками або embeddings від нейронних мереж.

Приклади застосування

Приклад 1: Прогнозування споживання електроенергії

Енергетична компанія використовує Gradient Boosting для прогнозування навантаження на електромережу. Модель враховує: - Історичні дані споживання (почасові за останні 5 років) - Метеорологічні дані: температура, вологість, швидкість вітру - Календарні ознаки: день тижня, святкові дні, сезон - Економічні показники: промислова активність, ціни на енергоносії

Система на основі LightGBM з 1000 дерев глибини 8 досягає середньої абсолютної похибки 2.3% для прогнозу на добу вперед. Аналіз важливості показує, що температура (35%), година доби (28%) та день тижня (15%) є найважливішими факторами. Це дозволяє оптимізувати роботу електростанцій та зменшити витрати на 12%.

Приклад 2: Детекція аномалій у виробництві

Автомобільний завод впроваджує систему контролю якості на основі Gradient Boosting. Система аналізує: - Дані з 200+ сенсорів: температура, тиск, вібрація, швидкість - Параметри виробничого процесу: швидкість конвеєра, налаштування обладнання - Історія технічного обслуговування - Візуальні інспекції та результати тестів

XGBoost модель з 500 деревами виявляє 94% дефектів на ранній стадії, що дозволяє: - Зменшити кількість бракованої продукції на 40% - Скоротити незаплановані простої на 25% - Оптимізувати графік технічного обслуговування

Приклад 3: Персоналізація цін в e-commerce

Онлайн-ритейлер використовує Gradient Boosting для динамічного ціноутворення: - Характеристики користувача: історія покупок, середній чек, частота візитів - Контекст: час доби, день тижня, сезон, погода - Конкурентне середовище: ціни конкурентів, наявність товару - Характеристики товару: категорія, бренд, новизна

CatBoost модель оптимізує ціни в реальному часі, максимізуючи прибуток з урахуванням еластичності попиту. Результати: - Збільшення виручки на 18% - Покращення конверсії на 23% - Оптимізація залишків на складі

Переваги та недоліки

Переваги

Висока точність прогнозування - Gradient Boosting часто перевершує інші алгоритми на структурованих даних. Послідовна оптимізація дозволяє досягти дуже низької помилки на навчальній вибірці з хорошим узагальненням.

Гнучкість у виборі функції втрат - можливість використання різних функцій втрат дозволяє адаптувати алгоритм до специфіки задачі: робастність до викидів, асиметричні втрати, квантильна регресія.

Обробка різнотипних даних - ефективно працює з числовими та категоріальними ознаками, автоматично моделює взаємодії між ознаками через глибину дерев.

Інтерпретованість - хоча складніша за Random Forest, модель все ще надає оцінки важливості ознак та можливість аналізу через SHAP values.

Ефективні реалізації - сучасні бібліотеки (XGBoost, LightGBM, CatBoost) високо оптимізовані, підтримують паралелізацію, GPU та розподілені обчислення.

Вбудована регуляризація - множина параметрів для контролю складності моделі: навчальна швидкість, глибина дерев, subsampling, регуляризація листків.

Обробка пропущених значень - сучасні реалізації ефективно працюють з пропущеними даними без необхідності попередньої імпутації.

Недоліки

Схильність до перенавчання - без правильної регуляризації може ідеально запам’ятати навчальні дані. Вимагає ретельного налаштування гіперпараметрів.

Послідовна природа - на відміну від Random Forest, дерева будуються послідовно, що ускладнює паралелізацію процесу навчання.

Чутливість до викидів - особливо при використанні квадратичної функції втрат. Викиди можуть сильно вплинути на градієнти та спотворити модель.

Довгий час навчання - для досягнення оптимальної якості часто потрібні тисячі ітерацій, що може займати значний час на великих датасетах.

Складність налаштування - велика кількість гіперпараметрів (learning rate, кількість дерев, глибина, subsampling, regularization) вимагає досвіду та обчислювальних ресурсів для оптимізації.

Потреба в великих обсягах даних - для ефективної роботи зазвичай потрібні тисячі прикладів, особливо при великій кількості ознак.

Екстраполяція - як і інші tree-based методи, не може прогнозувати за межами діапазону навчальних даних.

Нестабільність - невеликі зміни в даних або випадковості можуть призвести до суттєво різних моделей, особливо на початкових ітераціях.

Контрольні запитання

1. Поясніть основну відмінність між Random Forest та Gradient Boosting з точки зору побудови ансамблю.

2. Як градієнтний спуск у функціональному просторі пов’язаний з побудовою дерев у Gradient Boosting?

3. Чому для регресії з квадратичною функцією втрат градієнти дорівнюють залишкам?

4. Яка роль навчальної швидкості (shrinkage) у запобіганні перенавчанню?

5. Поясніть, як Gradient Boosting адаптується для багатокласової класифікації.

6. Чому використання неглибоких дерев часто дає кращі результати в Gradient Boosting?

7. Як стохастичний Gradient Boosting покращує узагальнюючу здатність моделі?

8. Опишіть процес знаходження оптимального кроку γ на кожній ітерації.

9. Які переваги має використання різних функцій втрат для різних задач?

10. Як можна оцінити важливість ознак у Gradient Boosting моделі?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму Gradient Boosting для задач регресії та класифікації

2. Мета

Зрозуміти принципи послідовної оптимізації в ансамблевих методах, навчитися реалізовувати Gradient Boosting з різними функціями втрат, освоїти техніки регуляризації.

3. Завдання

- Реалізувати базовий Gradient Boosting для регресії

- Додати підтримку різних функцій втрат

- Реалізувати версію для бінарної класифікації

- Додати техніки регуляризації (shrinkage, subsampling)

- Порівняти з існуючими реалізаціями

4. Опис

Студенти створюють повну реалізацію Gradient Boosting, починаючи з простої версії для регресії з квадратичною втратою, поступово додаючи підтримку інших функцій втрат, класифікації та методів регуляризації. Особлива увага приділяється розумінню ролі градієнтів та впливу різних гіперпараметрів на якість моделі.

5. Порядок виконання

1. Реалізувати обчислення градієнтів для різних функцій втрат

2. Створити базовий цикл Gradient Boosting для регресії

3. Реалізувати пошук оптимального кроку для кожної ітерації

4. Додати shrinkage та контроль глибини дерев

5. Реалізувати версію для бінарної класифікації з логістичною втратою

6. Додати стохастичний subsampling

7. Створити візуалізацію процесу навчання

8. Провести експерименти з різними конфігураціями

6. Результат

Робоча реалізація Gradient Boosting з підтримкою різних функцій втрат, методів регуляризації та можливістю візуалізації процесу оптимізації.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас GradientBoosting з методами fit та predict

- Функції обчислення градієнтів для різних втрат

- Метод оптимізації кроку на кожній ітерації
- Реалізація технік регуляризації
- Окремі версії для регресії та класифікації

2. Продемонструвати роботу додатку:
- Навчання моделі з візуалізацією зменшення помилки
- Вплив learning rate на швидкість збіжності
- Порівняння різних функцій втрат на одних даних
- Ефект від subsampling на якість моделі
- Аналіз важливості ознак

3. Бути готовим відповісти на питання:
- Як вибір функції втрат впливає на робастність до викидів?
- Чому оптимальний крок важливий для збіжності?
- Як глибина дерев впливає на bias-variance tradeoff?
- В чому переваги line search порівняно з фіксованим кроком?

Тести

1. Gradient Boosting оптимізує модель шляхом:
1) Паралельної побудови незалежних дерев
2) Послідовного додавання дерев, що виправляють помилки
3) Випадкового вибору найкращих дерев
4) Усереднення всіх можливих дерев

2. Псевдо-залишки в Gradient Boosting це:
1) Різниця між прогнозами різних дерев
2) Випадкові числа для регуляризації
3) Негативні градієнти функції втрат
4) Помилки на тестовій вибірці

3. Навчальна швидкість (learning rate) в Gradient Boosting:
1) Визначає швидкість росту дерев
2) Масштабує внесок кожного дерева
3) Контролює глибину дерев
4) Задає кількість ітерацій

4. Для квадратичної функції втрат градієнт дорівнює:

1) Квадрату помилки

2) Абсолютній помилці

3) Залишку (різниці між справжнім та прогнозованим)

4) Логарифму помилки

5. Stochastic Gradient Boosting використовує:

1) Випадкові функції втрат

2) Підвибірку даних на кожній ітерації

3) Випадкові дерева замість регресійних

4) Стохастичні градієнти

Відповіді:

1-2,

2-3,

3-2,

4-3,

5-2

Тести підвищеної складності

1. При використанні Gradient Boosting для квантильної регресії, градієнти залежать від:

1) Тільки величини помилки

2) Знаку помилки та заданого квантиля

3) Дисперсії прогнозів

4) Кількості дерев

2. Second-order методи (як у XGBoost) використовують не тільки градієнт, але й:

1) Третю похідну функції втрат

2) Градієнт попередньої ітерації

3) Гессіан (другу похідну)

4) Інтеграл функції втрат

3. Чому Gradient Boosting з експоненційною втратою еквівалентний AdaBoost?

1) Вони використовують однакові дерева

2) Градієнти експоненційної втрати призводять до тих самих ваг

3) Це різні алгоритми без зв'язку

4) AdaBoost є окремим випадком Random Forest

4. Для уникнення numerical overflow при softmax в багатокласовій класифікації використовують:

1) Логарифмування всіх значень

2) Віднімання максимального score перед експонентою

3) Обмеження глибини дерев

4) Нормалізацію градієнтів

5. LightGBM досягає швидкості через:

1) Використання GPU для всіх операцій

2) Gradient-based One-Side Sampling та Exclusive Feature Bundling

3) Зменшення кількості дерев

4) Спрощення функції втрат

6. При dart mode в LightGBM випадково "відкидаються" дерева для:

1) Прискорення навчання

2) Зменшення overfitting через dropout-подібний механізм

3) Економії пам'яті

4) Покращення інтерпретованості

7. CatBoost вирішує проблему target leakage для категоріальних ознак через:

1) One-hot encoding всіх категорій

2) Ordered target statistics

3) Ігнорування категоріальних ознак

4) Random encoding

8. Чому ранні stopping based на валідаційній помилці може бути проблематичним?

1) Валідаційна помилка завжди зменшується

2) Може зупинитися в локальному мінімумі через шум

3) Вимагає занадто багато даних

4) Не працює з shrinkage

9. Monotonic constraints в Gradient Boosting забезпечують:

1) Монотонне зменшення помилки

2) Монотонну залежність прогнозу від певних ознак

3) Однакову глибину всіх дерев

4) Постійну швидкість навчання

10. Feature interaction constraints обмежують:

1) Кількість використаних ознак

2) Які ознаки можуть з’являтися разом в одному дереві

3) Глибину взаємодій між ознаками

4) Важливість окремих ознак

Відповіді:

1. 2 - Градієнт залежить від того, чи помилка вище чи нижче нуля, масштабований квантилем

2. 3 - Другі похідні дозволяють краще апроксимувати функцію втрат

3. 2 - Математично градієнти призводять до ідентичних ваг прикладів як в AdaBoost

4. 2 - Log-sum-exp trick запобігає переповненню при збереженні правильних ймовірностей

5. 2 - GOSS зменшує кількість прикладів, EFB зменшує кількість ознак

6. 2 - Dropout дерев зменшує залежність від конкретних дерев

7. 2 - Використання впорядкованих статистик запобігає витоку інформації

8. 2 - Шумні коливання можуть призвести до передчасної зупинки

9. 2 - Гарантує, що збільшення ознаки не зменшить прогноз

10. 2 - Дозволяє контролювати складність моделі через обмеження взаємодій

Завдання для самостійної роботи

1. Реалізуйте Gradient Boosting з підтримкою custom функцій втрат, заданих користувачем через градієнт та гессіан.

2. Створіть адаптивну версію, яка автоматично вибирає learning rate на основі прогресу на валідаційній вибірці.

3. Реалізуйте multi-output Gradient Boosting для одночасного прогнозування кількох цільових змінних.

4. Розробіть метод візуалізації внеску кожної ітерації в фінальний прогноз для конкретного прикладу.

5. Імплементуйте Gradient Boosting з різними базовими учнями (лінійні моделі, GAM, невеликі нейронні мережі).

6. Створіть систему автоматичного feature engineering на основі аналізу розщеплень у побудованих деревах.

7. Реалізуйте методи інтерпретації моделі: partial dependence plots, SHAP values, feature interactions.

8. Розробіть стратегію ініціалізації F_0 на основі простішої моделі (наприклад, лінійної регресії).

9. Імплементуйте Gradient Boosting для survival analysis з підтримкою цензурованих даних.

10. Створіть distributed версію алгоритму для навчання на кластері з ефективною агрегацією градієнтів.

Теми лабораторних робіт

1. Базова реалізація Gradient Boosting для регресії з різними функціями втрат та візуалізацією збіжності.

2. Розробка фреймворку для порівняння впливу гіперпараметрів на bias-variance decomposition.

3. Імплементація Gradient Boosting для ранжування (Learning to Rank) з парними та списковими втратами.

4. Створення системи AutoML для автоматичного підбору оптимальної конфігурації Gradient Boosting.

5. Реалізація online Gradient Boosting для потокових даних з адаптацією до concept drift.

6. Розробка interpretable Gradient Boosting з автоматичною генерацією правил з ансамблю.

7. Порівняльний аналіз різних стратегій обробки категоріальних ознак (one-hot, target encoding, CatBoost-style).

8. Імплементація Gradient Boosting з диференційною приватністю для чутливих даних.

9. Створення гібридної моделі, що комбінує Gradient Boosting з deep learning для табличних даних.

10. Розробка системи мультимодального Gradient Boosting для одночасної роботи з різними типами даних.

Класифікація: SVM

Теоретичні відомості

Метод опорних векторів (Support Vector Machine, SVM) є одним з найпотужніших алгоритмів машинного навчання для задач класифікації та регресії. Розроблений Володимиром Вапніком та його колегами в 1990-х роках, SVM базується на теорії статистичного навчання та принципі структурної мінімізації ризику. Основна ідея алгоритму полягає в знаходженні оптимальної розділяючої гіперплощини в просторі ознак, яка максимізує відстань до найближчих точок різних класів.

Геометрична інтерпретація

Розглянемо задачу бінарної класифікації, де маємо навчальну вибірку $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, де $x_i \in \mathbb{R}^d$ - вектор ознак, а $y_i \in \{-1, +1\}$ - мітка класу. SVM шукає гіперплощину, яка розділяє точки двох класів з максимальним зазором (margin).

Гіперплощина у d-вимірному просторі описується рівнянням:

$$w^T x + b = 0$$

де $w \in \mathbb{R}^d$ - вектор нормалі до гіперплощини, $b \in \mathbb{R}$ - зсув (bias). Для класифікації нового прикладу x використовується знак функції рішення:

$$f(x) = \text{sign}(w^T x + b)$$

Максимізація зазору

Відстань від точки x_i до гіперплощини обчислюється як:

$$\frac{|w^T x_i + b|}{||w||}$$

де $||w|| = \sqrt{w^T w}$ - евклідова норма вектора w. Для правильно класифікованих точок маємо:

$$y_i(w^T x_i + b) > 0$$

Зазор (margin) визначається як мінімальна відстань від гіперплощини до найближчих точок обох класів. Ці найближчі точки називаються опорними векторами. Для спрощення математичних викладок вводиться канонічна форма гіперплощини, де для опорних векторів виконується:

$$y_i(w^T x_i + b) = 1$$

При такій нормалізації геометричний зазор дорівнює $\frac{2}{||w||}$. Максимізація зазору еквівалентна мінімізації $||w||^2$.

Формулювання задачі оптимізації

1. Жорсткий зазор (Hard Margin)

Для лінійно роздільних даних задача SVM формулюється як:

$$\min_{w,b} \frac{1}{2}||w||^2$$

за умов:

$$y_i(w^T x_i + b) \geq 1, \quad i = 1, \ldots, n$$

Це задача квадратичного програмування з лінійними обмеженнями.

2. М'який зазор (Soft Margin)

Для реальних даних, які можуть містити шум або не бути лінійно роздільними, вводяться змінні послаблення $\xi_i \geq 0$:

$$\min_{w,b,\xi} \frac{1}{2}||w||^2 + C\sum_{i=1}^{n} \xi_i$$

за умов:

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \quad i = 1, \ldots, n$$
$$\xi_i \geq 0, \quad i = 1, \ldots, n$$

Параметр $C > 0$ контролює баланс між максимізацією зазору та мінімізацією помилок класифікації.

Двоїста задача

Використовуючи метод множників Лагранжа, отримуємо двоїсту задачу:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

за умов:

$$0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n$$

$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

де α_i - множники Лагранжа. Рішення прямої задачі виражається через рішення двоїстої:

$$w = \sum_{i=1}^{n} \alpha_i y_i x_i$$

Опорними векторами є точки з $\alpha_i > 0$. Зсув b обчислюється з умов Каруша-Куна-Такера (ККТ).

Ядровий трюк (Kernel Trick)

Для нелінійно роздільних даних SVM використовує відображення у простір вищої розмірності через ядрову функцію. Ключова ідея полягає в тому, що в двоїстій задачі дані з'являються тільки у вигляді скалярних добутків $x_i^T x_j$, які можна замінити на ядрову функцію $K(x_i, x_j)$.

Популярні ядрові функції:

- Лінійне ядро: $K(x_i, x_j) = x_i^T x_j$

- Поліноміальне ядро: $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d$

- RBF (Radial Basis Function) ядро: $K(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2)$

- Сигмоїдне ядро: $K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r)$

Функція рішення з ядром:

$$f(x) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b\right)$$

Алгоритм SMO (Sequential Minimal Optimization)

SMO є ефективним алгоритмом для розв'язання двоїстої задачі SVM. Основна ідея полягає в ітеративній оптимізації пар множників Лагранжа при фіксованих інших.

Основні кроки SMO:

1. На кожній ітерації вибираються два множники α_i та α_j для оптимізації

2. Решта множників фіксуються, задача зводиться до одновимірної оптимізації

3. Аналітично знаходяться нові значення α_i та α_j

4. Оновлюється порогове значення b

5. Процес повторюється до збіжності

Вибір робочої пари

Ефективність SMO залежить від стратегії вибору пари множників:

Перший множник вибирається серед тих, що порушують умови ККТ: - Якщо $\alpha_i = 0$, то $y_i f(x_i) \geq 1$ - Якщо $0 < \alpha_i < C$, то $y_i f(x_i) = 1$ - Якщо $\alpha_i = C$, то $y_i f(x_i) \leq 1$

Другий множник вибирається для максимізації кроку оптимізації, часто використовується евристика максимальної різниці помилок $|E_i - E_j|$, де $E_k = f(x_k) - y_k$.

Псевдокод алгоритму SVM з SMO

```
функція НавчитиSVM(X, y, C, ядро, допуск, макс_ітерацій)
    // X - матриця навчальних прикладів
    // y - вектор міток класів {-1, +1}
    // C - параметр регуляризації
    // ядро - ядрова функція
    // допуск - точність збіжності
    // макс_ітерацій - максимальна кількість ітерацій

    n = кількість_прикладів(X)
    α = нулі(n)  // ініціалізація множників Лагранжа
    b = 0        // ініціалізація зсуву

    // Попередньо обчислити матрицю ядра
    K = матриця(n, n)
    для i = 1 до n виконати
        для j = 1 до n виконати
            K[i,j] = ядро(X[i], X[j])

    ітерація = 0
    поки ітерація < макс_ітерацій виконати
        кількість_змін = 0

        для i = 1 до n виконати
            E_i = ОбчислитиПомилку(i, α, b, X, y, K)

            якщо ПорушеноУмовиККТ(α[i], y[i], E_i, C, допуск) тоді
                // Вибрати другий множник
                j = ВибратиДругийМножник(i, E_i, α, X, y, K)
                якщо j == -1 тоді продовжити

                E_j = ОбчислитиПомилку(j, α, b, X, y, K)
```

```
                // Зберегти старі значення
                α_i_старе = α[i]
                α_j_старе = α[j]

                // Обчислити границі для α[j]
                L, H = ОбчислитиГраниці(α[i], α[j], y[i], y[j], C)
                якщо L == H тоді продовжити

                // Обчислити оптимальне значення α[j]
                η = 2*K[i,j] - K[i,i] - K[j,j]
                якщо η >= 0 тоді продовжити

                α[j] = α[j] - y[j]*(E_i - E_j)/η
                α[j] = ОбмежитиЗначення(α[j], L, H)

                якщо |α[j] - α_j_старе| < 1e-5 тоді продовжити

                // Оновити α[i]
                α[i] = α[i] + y[i]*y[j]*(α_j_старе - α[j])

                // Оновити поріг b
                b = ОновитиПоріг(E_i, E_j, α[i], α[j], α_i_старе, α_j
_старе,
                                 y[i], y[j], K[i,i], K[i,j], K[j,j],
b)

                кількість_змін += 1

        якщо кількість_змін == 0 тоді
            ітерація += 1
        інакше
            ітерація = 0

    // Знайти опорні вектори
    опорні_вектори = []
    для i = 1 до n виконати
        якщо α[i] > 0 тоді
            опорні_вектори.додати(i)

    повернути α, b, опорні_вектори

функція ОбчислитиПомилку(i, α, b, X, y, K)
    f_i = b
```

```
    для j = 1 до довжина(α) виконати
        якщо α[j] > 0 тоді
            f_i += α[j] * y[j] * K[i,j]
    повернути f_i - y[i]

функція ПорушеноУмовиККТ(α_i, y_i, E_i, C, допуск)
    якщо (y_i*E_i < -допуск і α_i < C) або (y_i*E_i > допуск і α_i >
0) тоді
        повернути істина
    повернути хиба

функція ВибратиДругийМножник(i, E_i, α, X, y, K)
    макс_крок = 0
    j = -1

    для k = 1 до довжина(α) виконати
        якщо k == i тоді продовжити

        E_k = ОбчислитиПомилку(k, α, b, X, y, K)
        крок = |E_i - E_k|

        якщо крок > макс_крок тоді
            макс_крок = крок
            j = k

    повернути j

функція ОбчислитиГраниці(α_i, α_j, y_i, y_j, C)
    якщо y_i != y_j тоді
        L = макс(0, α_j - α_i)
        H = мін(C, C + α_j - α_i)
    інакше
        L = макс(0, α_i + α_j - C)
        H = мін(C, α_i + α_j)
    повернути L, H

функція Класифікувати(x, α, b, X_train, y_train, ядро, опорні_вектори
)
    f = b
    для i в опорні_вектори виконати
        f += α[i] * y_train[i] * ядро(X_train[i], x)
    повернути sign(f)
```

Багатокласова класифікація

SVM природно розроблений для бінарної класифікації. Для багатокласових задач використовуються стратегії:

1. One-vs-All (OvA): Навчається k бінарних класифікаторів, кожен розділяє один клас від решти.

```
функція SVMБагатокласова_OvA(X, y, класи, C, ядро)
    моделі = []

    для клас в класи виконати
        // Створити бінарні мітки
        y_бінарні = []
        для i = 1 до довжина(y) виконати
            якщо y[i] == клас тоді
                y_бінарні.додати(+1)
            інакше
                y_бінарні.додати(-1)

        // Навчити бінарний SVM
        α, b, sv = НавчитиSVM(X, y_бінарні, C, ядро)
        моделі.додати((α, b, sv, клас))

    повернути моделі

функція КласифікуватиБагатокласовий(x, моделі, X_train, y_train, ядро
)
    макс_рішення = -нескінченність
    прогноз = null

    для (α, b, sv, клас) в моделі виконати
        рішення = b
        для i в sv виконати
            рішення += α[i] * y_train[i] * ядро(X_train[i], x)

        якщо рішення > макс_рішення тоді
            макс_рішення = рішення
            прогноз = клас

    повернути прогноз
```

2. One-vs-One (OvO): Навчається $\frac{k(k-1)}{2}$ бінарних класифікаторів для кожної пари класів.

Регуляризація та вибір параметрів

Продуктивність SVM суттєво залежить від вибору параметрів: - Параметр регуляризації C - Параметри ядра (наприклад, γ для RBF)

Оптимальні параметри зазвичай знаходяться через крос-валідацію з пошуком по сітці або байєсівською оптимізацією.

Приклад застосування

Розглянемо детальний приклад використання SVM для класифікації точок на площині. Маємо навчальний набір з 6 точок двох класів:

Клас +1: (2, 3), (3, 3), (3, 2)

Клас -1: (1, 1), (2, 1), (1, 2)

Крок 1: Ініціалізація

- Множники Лагранжа: $\alpha = [0,0,0,0,0,0]$

- Зсув: $b = 0$

- Параметр регуляризації: $C = 10$

- Використовуємо лінійне ядро: $K(x_i, x_j) = x_i^T x_j$

Крок 2: Обчислення матриці ядра

```
K = [[13, 15, 12, 5, 7, 5],
     [15, 18, 15, 6, 9, 6],
     [12, 15, 13, 5, 8, 5],
     [5,  6,  5,  2, 3, 2],
     [7,  9,  8,  3, 5, 3],
     [5,  6,  5,  2, 3, 2]]
```

Крок 3: Ітерація SMO

Ітерація 1:

- Обираємо $i = 1$ (точка (2,3), клас +1)

- Помилка: $E_1 = 0 - 1 = -1$

- Порушено умови ККТ: так ($y_1 E_1 = -1 < 0$ і $\alpha_1 = 0 < C$)

- Обираємо $j = 4$ (максимальна різниця помилок)

- Оновлюємо α_1 та α_4

Продовження ітерацій...

Крок 4: Результат Після збіжності отримуємо:

- Ненульові множники: $\alpha_2 = 2.5$, $\alpha_5 = 2.5$

- Опорні вектори: (3, 3) та (2, 1)

- Зсув: $b = -4$

Крок 5: Розділяюча гіперплощина

$$w = 2.5 \cdot (+1) \cdot \begin{pmatrix} 3 \\ 3 \end{pmatrix} + 2.5 \cdot (-1) \cdot \begin{pmatrix} 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 2.5 \\ 5 \end{pmatrix}$$

Рівняння гіперплощини: $2.5x_1 + 5x_2 - 4 = 0$

Крок 6: Класифікація нової точки Для точки (2.5, 2.5):

$$f(2.5,2.5) = 2.5 \cdot 2.5 + 5 \cdot 2.5 - 4 = 14.75 > 0$$

Прогноз: клас +1

Область застосування

SVM успішно застосовується в багатьох галузях завдяки своїй високій узагальнюючій здатності та теоретичним гарантіям:

Розпізнавання тексту та зображень - SVM показує відмінні результати в задачах оптичного розпізнавання символів (OCR), класифікації рукописних цифр, розпізнавання облич. Здатність працювати з високорозмірними даними робить його ідеальним для обробки зображень.

Біоінформатика - класифікація білків, прогнозування структури генів, аналіз мікрочіпів. SVM ефективно працює з малими навчальними вибірками при великій кількості ознак, що типово для біологічних даних.

Обробка природної мови - класифікація документів, визначення тональності тексту, розпізнавання іменованих сутностей. Ядрові функції дозволяють ефективно працювати з текстовими представленнями.

Фінансове прогнозування - оцінка кредитних ризиків, прогнозування банкрутства, виявлення шахрайства. Робастність до викидів та здатність моделювати нелінійні залежності цінні для фінансових даних.

Медична діагностика - класифікація захворювань за симптомами, аналіз медичних зображень, прогнозування ефективності лікування. Висока точність критична для медичних застосувань.

Детекція аномалій - One-class SVM використовується для виявлення аномалій у промисловості, кібербезпеці, моніторингу систем.

Приклади застосування

Приклад 1: Діагностика раку за мамографічними знімками

Медичний центр використовує SVM для автоматичного виявлення злоякісних утворень на мамограмах.

Система аналізує:

- Текстурні характеристики: контраст, кореляція, енергія, однорідність
- Геометричні параметри: площа, периметр, компактність утворень
- Інтенсивність та розподіл яскравості

SVM з RBF ядром ($\gamma = 0.1$, $C = 100$) навчається на 500 розмічених знімках. Модель досягає:

- Чутливість: 96% (правильно виявляє злоякісні утворення)
- Специфічність: 94% (правильно ідентифікує доброякісні)
- AUC: 0.98

Аналіз опорних векторів показує, що найважливішими є приклади на межі між класами - підозрілі випадки, які важко класифікувати навіть експертам.

Приклад 2: Фільтрація спаму в електронній пошті

Поштовий сервіс впроваджує SVM для класифікації вхідних повідомлень. Ознаки включають:

- Частоти ключових слів (TF-IDF представлення)
- Наявність специфічних фраз (“безкоштовно”, “виграш”, “терміново”)
- Характеристики відправника та заголовків
- Структурні особливості: кількість посилань, зображень, великих літер

Лінійний SVM з $C = 1$ навчається на 10000 розмічених листів.

Результати:

- Точність: 98.5% - False Positive Rate: 0.1% (важливі листи рідко потрапляють у спам)
- Швидкість класифікації: 0.1 мс на лист

Модель адаптується до нових типів спаму через періодичне донавчання на нових прикладах.

Приклад 3: Прогнозування відмов промислового обладнання

Виробниче підприємство використовує SVM для predictive maintenance. Модель аналізує:

- Дані з сенсорів: температура, вібрація, тиск, швидкість обертання - Історію обслуговування та ремонтів

- Умови експлуатації: навантаження, тривалість роботи

- Акустичні сигнали від обладнання

One-class SVM з RBF ядром навчається на даних нормальної роботи. Система:

- Виявляє 92% відмов за 48 годин до критичної поломки
- Знижує незаплановані простої на 35%
- Оптимізує графік технічного обслуговування

Візуалізація в просторі перших двох головних компонент показує чітке відділення аномальних станів від нормальних.

Переваги та недоліки

Переваги

Теоретичні гарантії - SVM базується на солідній математичній теорії (теорія Вапніка-Червоненкіса), що забезпечує гарантії узагальнення. Принцип структурної мінімізації ризику дозволяє контролювати складність моделі.

Ефективність у високорозмірних просторах - SVM добре працює навіть коли кількість ознак перевищує кількість прикладів, що типово для текстових даних та біоінформатики.

Гнучкість через ядрові функції - можливість неявно працювати в просторах дуже високої (навіть нескінченної) розмірності дозволяє моделювати складні нелінійні залежності.

Робастність до перенавчання - максимізація зазору природно регуляризує модель, особливо ефективно при правильному виборі параметра C.

Глобальний оптимум - опукла задача оптимізації гарантує знаходження глобального, а не локального оптимуму.

Розріджене рішення - модель визначається лише опорними векторами, що робить її компактною та швидкою для прогнозування.

Універсальність - один алгоритм працює для класифікації, регресії та виявлення аномалій з мінімальними модифікаціями.

Недоліки

Чутливість до вибору параметрів - продуктивність сильно залежить від вибору ядра та його параметрів, параметра регуляризації C. Потрібен ретельний підбір через крос-валідацію.

Обчислювальна складність - навчання має складність $O(n^2)$ до $O(n^3)$, що робить його повільним для великих датасетів. Зберігання матриці ядра вимагає $O(n^2)$ пам'яті.

Складність інтерпретації - нелінійні SVM важко інтерпретувати, на відміну від лінійних моделей чи дерев рішень. Важко зрозуміти, чому модель прийняла конкретне рішення.

Відсутність прямих оцінок ймовірностей - SVM дає лише бінарні рішення або відстані до гіперплощини. Для отримання ймовірностей потрібні додаткові методи (Platt scaling).

Проблеми з незбалансованими даними - стандартний SVM може бути зміщений у бік мажоритарного класу. Потрібні модифікації (class weights) або спеціальні підходи.

Вибір ядра - немає універсального правила вибору ядрової функції для конкретної задачі. Часто потрібні експерименти з різними ядрами.

Масштабування для багатокласових задач - стратегії OvA та OvO вимагають навчання багатьох моделей, що збільшує час навчання та прогнозування.

Контрольні запитання

1. Поясніть геометричний зміст максимізації зазору в SVM та чому це покращує узагальнюючу здатність.

2. Яка роль опорних векторів у побудові розділяючої гіперплощини та чому інші точки не впливають на рішення?

3. Поясніть різницю між жорстким та м'яким зазором. В яких випадках використовується кожен підхід?

4. Як ядровий трюк дозволяє SVM працювати з нелінійно роздільними даними без явного відображення у простір вищої розмірності?

5. Опишіть умови Каруша-Куна-Такера для задачі SVM та їх роль в алгоритмі SMO.

6. Чому двоїста задача часто простіша для розв'язання, ніж пряма? Які переваги це дає?

7. Як параметр регуляризації C впливає на баланс між точністю на навчальній вибірці та складністю моделі?

8. Поясніть стратегії One-vs-All та One-vs-One для багатокласової класифікації. Які їх переваги та недоліки?

9. Чому RBF ядро є універсальним вибором для багатьох задач? Як параметр γ впливає на гнучкість моделі?

10. Як можна адаптувати SVM для задач з незбалансованими класами або різною ціною помилок?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація алгоритму SVM з різними ядровими функціями

2. Мета

Зрозуміти принципи роботи методу опорних векторів, навчитися реалізовувати алгоритм SMO, дослідити вплив різних ядер та параметрів на якість класифікації.

3. Завдання

- Реалізувати базовий лінійний SVM
- Додати підтримку різних ядрових функцій
- Реалізувати алгоритм SMO для оптимізації
- Додати можливість м’якого зазору
- Візуалізувати границі рішення та опорні вектори
- Реалізувати багатокласову класифікацію

4. Опис

Студенти розробляють повну реалізацію SVM, починаючи з простого випадку лінійно роздільних даних, поступово додаючи підтримку ядрових функцій, м’якого зазору та багатокласової класифікації. Особлива увага приділяється розумінню ролі опорних векторів та впливу параметрів на модель.

5. Порядок виконання

1. Реалізувати функції обчислення різних ядер (лінійне, поліноміальне, RBF)
2. Створити структури даних для зберігання моделі SVM
3. Реалізувати алгоритм SMO з евристиками вибору робочої пари
4. Додати обчислення порогу b з умов ККТ

5. Реалізувати функцію прогнозування для нових точок

6. Створити візуалізацію для 2D даних

7. Реалізувати стратегії OvA та OvO для багатокласової класифікації

8. Провести експерименти з різними наборами даних

6. Результат

Функціональна реалізація SVM з можливістю вибору ядра, візуалізацією процесу навчання та границь рішення, підтримкою багатокласової класифікації.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас SVM з методами fit та predict

- Реалізація алгоритму SMO

- Функції різних ядер

- Методи для обчислення та оновлення параметрів

- Візуалізація границь рішення

- Реалізація багатокласових стратегій

2. Продемонструвати роботу додатку:

- Навчання SVM на лінійно роздільних даних

- Використання різних ядер для нелінійних даних

- Вплив параметрів C та γ на границі рішення

- Ідентифікація опорних векторів

- Порівняння OvA та OvO стратегій

3. Бути готовим відповісти на питання:

- Як змінюється кількість опорних векторів при зміні C?

- Чому деякі точки стають опорними векторами?

- Як вибір ядра впливає на форму границі рішення?

- Яка роль евристик у виборі робочої пари в SMO?

Тести

1. Опорними векторами в SVM є:

1) Всі навчальні приклади

2) Точки, найближчі до розділяючої гіперплощини

3) Точки з найбільшими вагами

4) Випадково вибрані точки

2. Ядровий трюк дозволяє:

1) Прискорити навчання SVM

2) Працювати в просторі високої розмірності без явного відображення

3) Зменшити кількість опорних векторів

4) Автоматично вибрати оптимальні параметри

3. Параметр C в SVM контролює:

1) Кількість опорних векторів

2) Складність ядрової функції

3) Баланс між максимізацією зазору та мінімізацією помилок

4) Розмірність простору ознак

4. RBF ядро визначається як:

1) $K(x,y) = x^T y$

2) $K(x,y) = (x^T y + 1)^d$

3) $K(x,y) = \exp(-\gamma||x-y||^2)$

4) $K(x,y) = \tanh(\gamma x^T y)$

5. Алгоритм SMO оптимізує:

1) Всі множники Лагранжа одночасно

2) По одному множнику за ітерацію

3) Пари множників за ітерацію

4) Випадкову кількість множників

Відповіді:

1-2,

2-2,

3-3,

4-3,

5-3

Тести підвищеної складності

1. При використанні поліноміального ядра ступеня d, простір ознак має розмірність:

1) d

2) $\binom{n+d}{d}$ для n вихідних ознак

3) n × d

4) n^d

2. Умова complementary slackness в ККТ для SVM стверджує:

1) α_i = 0 або y_i(w^T x_i + b) = 1

2) α_i(y_i(w^T x_i + b) - 1 + ξ_i) = 0

3) Всі α_i повинні бути рівними

4) Сума всіх α_i дорівнює C

3. ν-SVM відрізняється від C-SVM тим, що:

1) Використовує інше ядро

2) Параметр ν контролює частку опорних векторів та помилок

3) Працює тільки для регресії

4) Не потребує оптимізації

4. Для отримання ймовірнісних оцінок з SVM використовується:

1) Softmax функція на виходах

2) Platt scaling з сигмоїдною функцією

3) Нормалізація відстаней до гіперплощини

4) Bootstrap оцінки

5. Working set selection в SMO найефективніше реалізується через:

1) Випадковий вибір пар

2) Максимізацію порушення ККТ та евристику максимального кроку

3) Перебір всіх можливих пар

4) Вибір пар з мінімальними α

6. Structural Risk Minimization в контексті SVM означає:

1) Мінімізацію емпіричного ризику

2) Баланс між емпіричним ризиком та складністю моделі

3) Максимізацію кількості опорних векторів

4) Використання найпростішого ядра

7. String kernels для текстових даних обчислюють:

1) Евклідову відстань між рядками

2) Кількість спільних підрядків з урахуванням розривів

3) Довжину найбільшої спільної підпослідовності

4) Хеш-суму рядків

8. Чому матриця Грама повинна бути позитивно напіввизначеною?

1) Для швидшої збіжності SMO

2) Щоб гарантувати опуклість задачі оптимізації

3) Для зменшення кількості опорних векторів

4) Це не обов’язкова вимога

9. Online SVM відрізняється від batch версії:

1) Використанням іншого ядра

2) Інкрементальним оновленням моделі при надходженні нових даних

3) Відсутністю опорних векторів

4) Лінійною складністю навчання

10. Transductive SVM використовує:

1) Тільки розмічені дані

2) Розмічені та нерозмічені дані для покращення границі

3) Синтетичні дані для augmentation

4) Transfer learning з інших задач

Відповіді:

1. 2 - Розмірність простору мономів ступеня $\leqslant d$ визначається біноміальним коефіцієнтом

2. 2 - Повна форма complementary slackness включає змінні послаблення

3. 2 - ν є верхньою межею частки помилок та нижньою межею частки SV

4. 2 - Platt scaling апроксимує posterior ймовірності сигмоїдою

5. 2 - Поєднання WSS1 (максимальне порушення) та WSS2 (максимальний крок)

6. 2 - SRM мінімізує верхню межу ризику узагальнення

7. 2 - String kernels враховують всі спільні підпослідовності з gap penalties

8. 2 - Теорема Мерсера гарантує опуклість для PSD ядер

9. 2 - Online SVM адаптує модель без повного перенавчання

10. 2 - TSVM використовує структуру нерозмічених даних для кращої границі

Завдання для самостійної роботи

1. Реалізуйте ієрархічний SVM для класифікації з великою кількістю класів, використовуючи таксономію класів.

2. Створіть адаптивну версію SVM, яка автоматично вибирає оптимальне ядро та параметри на основі характеристик даних.

3. Реалізуйте Least Squares SVM, який зводить задачу до розв'язання системи лінійних рівнянь.

4. Розробіть методи візуалізації високорозмірних границь рішення SVM через проекції та перетини.

5. Імплементуйте incremental SVM для потокових даних з обмеженою пам'яттю для опорних векторів.

6. Створіть ensemble з SVM моделей з різними ядрами та параметрами для покращення узагальнення.

7. Реалізуйте SVM для структурованого виводу (Structured SVM) для задач послідовного мічення.

8. Розробіть методи інтерпретації SVM моделей через decomposition границі рішення.

9. Імплементуйте multi-instance SVM для задач, де приклади групуються в мішки.

10. Створіть distributed SVM для навчання на великих датасетах з використанням MapReduce парадигми.

Теми лабораторних робіт

1. Базова реалізація лінійного та ядрового SVM з візуалізацією для 2D даних.

2. Розробка системи автоматичного підбору параметрів SVM з використанням крос-валідації та grid search.

3. Порівняльний аналіз різних ядрових функцій на задачах класифікації текстів та зображень.

4. Імплементація One-class SVM для виявлення аномалій у часових рядах.

5. Створення SVM-based системи розпізнавання рукописних символів з custom ядрами.

6. Реалізація Semi-supervised SVM з використанням нерозмічених даних.

7. Розробка Real-time SVM класифікатора з оптимізаціями для швидкого прогнозування.

8. Порівняння різних стратегій роботи з незбалансованими даними в SVM.

9. Імплементація Multiple Kernel Learning для автоматичної комбінації ядер.

10. Створення інтерактивної системи для дослідження впливу outliers на SVM модель.

Наївний байєсівський класифікатор

Теоретичні відомості

Наївний байєсівський класифікатор (Bayes) є ймовірнісним алгоритмом машинного навчання, який базується на теоремі Байєса з припущенням про умовну незалежність ознак. Незважаючи на свою простоту та "наївне" припущення про незалежність, цей алгоритм демонструє високу ефективність у багатьох практичних застосуваннях, особливо в задачах обробки природної мови та класифікації документів.

Теорема Байєса

Основою алгоритму є теорема Байєса, яка дозволяє обчислити апостеріорну ймовірність події на основі апріорної ймовірності та правдоподібності:

$$P(C|X) = \frac{P(X|C) \cdot P(C)}{P(X)}$$

де: - $P(C|X)$ - апостеріорна ймовірність класу C за умови спостереження ознак X - $P(X|C)$ - правдоподібність (likelihood) спостереження ознак X за умови класу C - $P(C)$ - апріорна ймовірність класу C - $P(X)$ - ймовірність спостереження ознак X (evidence)

Для задачі класифікації нас цікавить клас з максимальною апостеріорною ймовірністю:

$$\hat{C} = \underset{c}{\operatorname{argmax}} P(C = c|X)$$

Оскільки $P(X)$ однакова для всіх класів, її можна не враховувати при порівнянні:

$$\hat{C} = \underset{c}{\operatorname{argmax}} P(X|C = c) \cdot P(C = c)$$

Ключовим припущенням Bayes є умовна незалежність ознак за даного класу. Це означає, що за умови належності до певного класу, ймовірність спостереження однієї ознаки не залежить від значень інших ознак.

Для вектора ознак $X = (x_1, x_2, \ldots, x_n)$ це припущення дозволяє розкласти правдоподібність:

$$P(X|C) = P(x_1, x_2, \ldots, x_n|C) = \prod_{i=1}^{n} P\,(x_i|C)$$

Таким чином, правило класифікації набуває вигляду:

$$\hat{C} = \underset{c}{\operatorname{argmax}} P(C = c) \prod_{i=1}^{n} P\,(x_i|C = c)$$

Типи Bayes класифікаторів

Залежно від природи ознак та припущень про їх розподіл, розрізняють кілька варіантів алгоритму:

1. Гаусівський Bayes використовується для неперервних ознак з припущенням про нормальний розподіл:

$$P(x_i|C) = \frac{1}{\sqrt{2\pi\sigma_{ic}^2}} \exp\left(-\frac{(x_i - \mu_{ic})^2}{2\sigma_{ic}^2}\right)$$

де μ_{ic} та σ_{ic}^2 - середнє та дисперсія ознаки i для класу c.

2. Мультиноміальний Bayes підходить для дискретних ознак, що представляють частоти або кількості (наприклад, частоти слів у документі):

$$P(x_i|C) = \frac{N_{ic} + \alpha}{N_c + \alpha n}$$

де N_{ic} - кількість появ ознаки i в класі c, N_c - загальна кількість ознак в класі c, α - параметр згладжування Лапласа, n - кількість унікальних ознак.

3. Бернуллівський Bayes працює з бінарними ознаками (наявність/відсутність):

$$P(x_i|C) = P(x_i = 1|C)^{x_i} \cdot (1 - P(x_i = 1|C))^{1-x_i}$$

Згладжування

Проблема нульових ймовірностей виникає, коли певна ознака не зустрічається в навчальних даних для деякого класу. Це призводить до нульової ймовірності всього добутку. Для вирішення цієї проблеми використовують техніки згладжування.

Згладжування Лапласа (add-one smoothing) додає псевдо-лічильник до всіх частот:

$$P(x_i|C) = \frac{count(x_i, C) + \alpha}{count(C) + \alpha \cdot |V|}$$

де $|V|$ - розмір словника (кількість унікальних значень ознаки), α - параметр згладжування (зазвичай 1).

Логарифмічний простір

Для запобігання арифметичному переповненню при множенні малих ймовірностей, обчислення часто виконуються в логарифмічному просторі:

$$\log P(C|X) = \log P(C) + \sum_{i=1}^{n} \log P(x_i|C)$$

Класифікація відбувається за правилом:

$$\hat{C} = \underset{c}{\operatorname{argmax}} \left[\log P(C = c) + \sum_{i=1}^{n} \log P(x_i|C = c) \right]$$

Псевдокод алгоритму Bayes

```
функція НавчитиNaiveBayes(X_train, y_train, тип_моделі, α)
    // X_train - матриця навчальних прикладів
    // y_train - вектор міток класів
    // тип_моделі - "gaussian", "multinomial", або "bernoulli"
    // α - параметр згладжування (для multinomial та bernoulli)

    класи = унікальні(y_train)
    n_класів = довжина(класи)
    n_ознак = кількість_стовпців(X_train)

    // Обчислити апріорні ймовірності
    апріорні = словник()
    для клас в класи виконати
        апріорні[клас] = кількість(y_train == клас) / довжина(y_train
)

    // Обчислити параметри умовних розподілів
    параметри = словник()
```

```
    якщо тип_моделі == "gaussian" тоді
        для клас в класи виконати
            X_клас = X_train[y_train == клас]
            параметри[клас] = словник()
            параметри[клас]["середні"] = середнє(X_клас, вісь=0)
            параметри[клас]["дисперсії"] = дисперсія(X_клас, вісь=0)

    якщо тип_моделі == "multinomial" тоді
        для клас в класи виконати
            X_клас = X_train[y_train == клас]
            параметри[клас] = словник()
            // Підрахунок частот з згладжуванням Лапласа
            частоти = сума(X_клас, вісь=0) + α
            загальна_кількість = сума(частоти)
            параметри[клас]["ймовірності"] = частоти / загальна_кільк
ість

    якщо тип_моделі == "bernoulli" тоді
        для клас в класи виконати
            X_клас = X_train[y_train == клас]
            параметри[клас] = словник()
            // Ймовірність наявності ознаки
            кількість_присутніх = сума(X_клас > 0, вісь=0) + α
            загальна_кількість = довжина(X_клас) + 2*α
            параметри[клас]["ймовірності"] = кількість_присутніх / за
гальна_кількість

    модель = словник()
    модель["класи"] = класи
    модель["апріорні"] = апріорні
    модель["параметри"] = параметри
    модель["тип"] = тип_моделі
    модель["α"] = α

    повернути модель

функція ПрогнозуватиNaiveBayes(модель, X_test)
    n_прикладів = кількість_рядків(X_test)
    прогнози = масив(n_прикладів)

    для i = 0 до n_прикладів-1 виконати
        x = X_test[i]
        макс_логарифм_ймовірність = -нескінченність
        кращий_клас = null
```

```
        для клас в модель["класи"] виконати
            // Логарифм апріорної ймовірності
            лог_ймовірність = log(модель["апріорні"][клас])

            // Додати логарифми умовних ймовірностей
            для j = 0 до довжина(x)-1 виконати
                умовна_ймовірність = ОбчислитиУмовнуЙмовірність(
                    x[j], j, клас, модель
                )
                якщо умовна_ймовірність > 0 тоді
                    лог_ймовірність += log(умовна_ймовірність)
                інакше
                    лог_ймовірність = -нескінченність
                    перервати

            якщо лог_ймовірність > макс_логарифм_ймовірність тоді
                макс_логарифм_ймовірність = лог_ймовірність
                кращий_клас = клас

        прогнози[i] = кращий_клас

    повернути прогнози

функція ОбчислитиУмовнуЙмовірність(значення, індекс_ознаки, клас, мод
ель)
    параметри_класу = модель["параметри"][клас]

    якщо модель["тип"] == "gaussian" тоді
        μ = параметри_класу["середні"][індекс_ознаки]
        σ² = параметри_класу["дисперсії"][індекс_ознаки]
        // Щільність нормального розподілу
        повернути (1/sqrt(2*π*σ²)) * exp(-(значення-μ)²/(2*σ²))

    якщо модель["тип"] == "multinomial" тоді
        повернути параметри_класу["ймовірності"][індекс_ознаки]

    якщо модель["тип"] == "bernoulli" тоді
        p = параметри_класу["ймовірності"][індекс_ознаки]
        якщо значення > 0 тоді
            повернути p
        інакше
            повернути 1 - p

// Функція для обчислення ймовірностей класів
```

```
функція ЙмовірностіКласів(модель, x)
    ймовірності = словник()
    сума_ймовірностей = 0

    для клас в модель["класи"] виконати
        лог_ймовірність = log(модель["апріорні"][клас])

        для j = 0 до довжина(x)-1 виконати
            умовна = ОбчислитиУмовнуЙмовірність(x[j], j, клас, модель
)
            якщо умовна > 0 тоді
                лог_ймовірність += log(умовна)

        ймовірності[клас] = exp(лог_ймовірність)
        сума_ймовірностей += ймовірності[клас]

    // Нормалізація
    для клас в модель["класи"] виконати
        ймовірності[клас] /= сума_ймовірностей

    повернути ймовірності
```

Обробка текстових даних

Для класифікації текстів потрібна попередня обробка:

```
функція ПідготуватиТекстовіДані(документи, макс_ознак=null)
    // Створити словник
    словник = множина()
    для документ в документи виконати
        слова = токенізувати(документ.нижній_регістр())
        словник.об'єднати(слова)

    // Обмежити розмір словника
    якщо макс_ознак != null і довжина(словник) > макс_ознак тоді
        // Залишити найчастіші слова
        частоти = підрахувати_частоти(документи)
        словник = топ_слова(частоти, макс_ознак)

    // Перетворити в індекси
    слово_до_індексу = словник()
    індекс = 0
    для слово в словник виконати
        слово_до_індексу[слово] = індекс
        індекс += 1

    // Векторизація документів
```

```
    X = матриця(довжина(документи), довжина(словник))

    для і, документ в перелічити(документи) виконати
        слова = токенізувати(документ.нижній_регістр())
        для слово в слова виконати
            якщо слово в слово_до_індексу тоді
                j = слово_до_індексу[слово]
                X[i][j] += 1  // Для multinomial
                // X[i][j] = 1  // Для bernoulli

    повернути X, слово_до_індексу
```

Приклад застосування

Розглянемо детальний приклад класифікації електронних листів на спам та не-спам використовуючи Bayes.

Навчальні дані:

Маємо 6 листів з відомими мітками:

1. “безкоштовно гроші зараз” → спам

2. “виграй приз сьогодні” → спам

3. “зустріч завтра офіс” → не-спам

4. “проект готовий завтра” → не-спам

5. “безкоштовно приз виграй” → спам

6. “офіс проект зустріч” → не-спам

Крок 1: Створення словника та векторизація

Словник: {безкоштовно, гроші, зараз, виграй, приз, сьогодні, зустріч, завтра, офіс, проект, готовий}

Матриця частот (bag-of-words):

```
         безк. гроші зараз вигр. приз сьог. зуст. завтра офіс прое. г
от.
Лист 1:    1     1     1     0     0     0     0      0     0     0
0
Лист 2:    0     0     0     1     1     1     0      0     0     0
0
Лист 3:    0     0     0     0     0     0     1      1     1     0
0
Лист 4:    0     0     0     0     0     0     0      1     0     1
1
Лист 5:    1     0     0     1     1     0     0      0     0     0
0
```

```
Лист 6:    0     0     0     0     0    0     1      0     1     1
0
```

Крок 2: Обчислення апріорних ймовірностей

- P(спам) = 3/6 = 0.5

- P(не-спам) = 3/6 = 0.5

Крок 3: Обчислення умовних ймовірностей (з згладжуванням Лапласа, α=1)

Для класу “спам” (листи 1, 2, 5):

- Загальна кількість слів: 8

- P(безкоштовно|спам) = (2+1)/(8+11) = 3/19

- P(гроші|спам) = (1+1)/(8+11) = 2/19

- P(виграй|спам) = (2+1)/(8+11) = 3/19

- P(приз|спам) = (2+1)/(8+11) = 3/19

- P(зустріч|спам) = (0+1)/(8+11) = 1/19 - ... (аналогічно для інших слів)

Для класу “не-спам” (листи 3, 4, 6):

- Загальна кількість слів: 8

- P(зустріч|не-спам) = (2+1)/(8+11) = 3/19

- P(офіс|не-спам) = (2+1)/(8+11) = 3/19

- P(проект|не-спам) = (2+1)/(8+11) = 3/19

- P(безкоштовно|не-спам) = (0+1)/(8+11) = 1/19 - ... (аналогічно для інших слів)

Крок 4: Класифікація нового листа

Новий лист: “безкоштовно офіс зустріч”

Обчислюємо логарифми ймовірностей:

Для класу “спам”:

```
log P(спам) + log P(безкоштовно|спам) + log P(офіс|спам) + log P(зуст
річ|спам)
= log(0.5) + log(3/19) + log(1/19) + log(1/19)
= -0.693 + (-1.846) + (-2.944) + (-2.944)
= -8.427
```

Для класу “не-спам”:

```
log P(не-спам) + log P(безкоштовно|не-спам) + log P(офіс|не-спам) + l
og P(зустріч|не-спам)
= log(0.5) + log(1/19) + log(3/19) + log(3/19)
```

```
= -0.693 + (-2.944) + (-1.846) + (-1.846)
= -7.329
```
Результат: -7.329 > -8.427, тому лист класифікується як “не-спам”.

Область застосування

Bayes класифікатор широко використовується завдяки своїй простоті, швидкості та ефективності:

Фільтрація спаму - один з найуспішніших застосувань, де алгоритм аналізує частоти слів для визначення спам-повідомлень. Швидкість навчання та класифікації робить його ідеальним для real-time фільтрації.

Аналіз тональності - визначення емоційного забарвлення текстів (позитивне, негативне, нейтральне) в соціальних мережах, відгуках користувачів, новинах. Мультиноміальний варіант ефективно працює з частотами слів.

Класифікація документів - автоматична категоризація новин, наукових статей, веб-сторінок. Алгоритм добре масштабується на великі колекції документів та багато категорій.

Медична діагностика - прогнозування захворювань на основі симптомів. Незважаючи на припущення про незалежність, часто дає точні результати для попереднього скринінгу.

Рекомендаційні системи - прогнозування вподобань користувачів на основі попередньої активності. Швидкість дозволяє робити рекомендації в реальному часі.

Розпізнавання образів - класифікація зображень, розпізнавання рукописного тексту. Гаусівський варіант працює з числовими ознаками зображень.

Приклади застосування

Приклад 1: Автоматична модерація коментарів

Соціальна платформа використовує Bayes для виявлення токсичних коментарів. Система аналізує: - Наявність образливих слів та фраз - Використання великих літер (CAPS LOCK) - Кількість знаків оклику та питання - N-грами (послідовності з 2-3 слів) - Емотикони та емодзі

Бернуллівський Bayes з словником 10,000 найчастіших токенів досягає: - Точність: 94% - Precision для токсичних коментарів: 89% - Recall: 92% - Час класифікації: < 1 мс на коментар

Система обробляє 1 мільйон коментарів на день, автоматично приховуючи потенційно проблемні для ручної перевірки.

Приклад 2: Діагностика захворювань щитовидної залози

Медична клініка використовує Гаусівський Bayes для попередньої діагностики. Модель враховує: - Рівні гормонів: TSH, T3, T4, FTI - Демографічні дані: вік, стать - Симптоми: втома, зміна ваги, серцебиття - Анамнез: вагітність, прийом ліків, операції

На датасеті з 7,200 пацієнтів модель розрізняє: - Норма - Гіпотиреоз - Гіпертиреоз

Результати: - Загальна точність: 96% - Чутливість для гіпертиреозу: 98% (критично для швидкого лікування) - Специфічність: 95%

Модель використовується для пріоритизації пацієнтів для детального обстеження.

Приклад 3: Персоналізація новинної стрічки

Новинний агрегатор використовує Bayes для прогнозування цікавості статей для користувача: - Аналізується історія кліків користувача - Витягуються ознаки статей: категорія, ключові слова, джерело, довжина - Враховується час публікації та день тижня

Мультиноміальний Bayes з TF-IDF вагами: - Навчається на 100,000 взаємодій користувача - Оновлюється щодня для адаптації до змін інтересів - CTR (Click-Through Rate) зростає на 35% - Час персоналізації: 50 мс на 1000 статей

Переваги та недоліки

Переваги

Простота та швидкість - алгоритм легко зрозуміти та реалізувати. Навчання зводиться до підрахунку частот, що робить його одним з найшвидших алгоритмів машинного навчання. Класифікація також дуже швидка - O(nk), де n - кількість ознак, k - кількість класів.

Ефективність на малих даних - на відміну від глибокого навчання, Bayes може давати хороші результати навіть на невеликих навчальних вибірках. Це особливо цінно в задачах з обмеженими розміченими даними.

Масштабованість - алгоритм легко масштабується на великі датасети та велику кількість ознак. Пам’ять, необхідна для зберігання моделі, лінійно залежить від кількості ознак та класів.

Інкрементальне навчання - модель легко оновлювати при надходженні нових даних без повного перенавчання. Достатньо оновити лічильники та перерахувати ймовірності.

Робота з високорозмірними даними - алгоритм природно працює з тисячами ознак, що робить його популярним для текстової класифікації.

Ймовірнісні прогнози - на відміну від деяких алгоритмів, Bayes дає оцінки ймовірностей належності до класів, що корисно для ранжування та прийняття рішень з урахуванням невизначеності.

Стійкість до нерелевантних ознак - завдяки множенню ймовірностей, нерелевантні ознаки з рівномірним розподілом мало впливають на результат.

Недоліки

Припущення про незалежність - основне обмеження алгоритму. В реальних даних ознаки часто корельовані, що може призводити до неточних оцінок ймовірностей. Наприклад, слова “машинне” та “навчання” часто з’являються разом.

Проблема нульових частот - якщо ознака не зустрічалась у навчальних даних для певного класу, це призводить до нульової ймовірності. Хоча згладжування частково вирішує проблему, воно вносить додаткове зміщення.

Чутливість до дисбалансу класів - при сильному дисбалансі апріорні ймовірності можуть домінувати над умовними, призводячи до зміщення в бік мажоритарного класу.

Неточні оцінки ймовірностей - хоча алгоритм часто дає правильну класифікацію, абсолютні значення ймовірностей можуть бути далекими від реальних через порушення припущення про незалежність.

Обмежена виразність - лінійна комбінація log-ймовірностей обмежує складність границь рішення. Алгоритм не може моделювати складні нелінійні залежності.

Проблеми з числовими ознаками - припущення про нормальний розподіл в Гаусівському варіанті часто не виконується. Дискретизація числових ознак може призводити до втрати інформації.

Погана робота з корельованими ознаками - дублювання інформації в корельованих ознаках може призводити до надмірної ваги певних аспектів даних.

Контрольні запитання

1. Поясніть, чому алгоритм називається “наївним” та як це припущення спрощує обчислення.

2. Як теорема Байєса використовується для класифікації та чому можна ігнорувати знаменник P(X)?

3. Опишіть різницю між Гаусівським, Мультиноміальним та Бернуллівським варіантами Bayes.

4. Яка проблема виникає при нульових частотах та як її вирішує згладжування Лапласа?

5. Чому обчислення часто виконуються в логарифмічному просторі?

6. Як Bayes обробляє неперервні ознаки та які припущення при цьому робляться?

7. Поясніть, чому Bayes часто працює добре на практиці, незважаючи на нереалістичне припущення про незалежність.

8. Як можна адаптувати Bayes для роботи з незбалансованими класами?

9. Які переваги має Bayes для текстової класифікації порівняно з іншими алгоритмами?

10. Як можна оцінити важливість ознак в моделі Bayes?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження Bayes класифікатора для різних типів даних

2. Мета

Зрозуміти принципи роботи байєсівської класифікації, навчитися реалізовувати різні варіанти Bayes, дослідити вплив припущення про незалежність на якість класифікації.

3. Завдання

- Реалізувати Гаусівський Bayes для числових даних

- Реалізувати Мультиноміальний Bayes для текстів

- Додати різні методи згладжування

- Реалізувати методи оцінки важливості ознак

- Порівняти з іншими алгоритмами класифікації

- Візуалізувати границі рішення та розподіли ймовірностей

4. Опис

Студенти створюють повну реалізацію різних варіантів Bayes, включаючи попередню обробку даних, навчання моделі, прогнозування та оцінку якості. Особлива увага приділяється розумінню впливу припущення про незалежність та методам його пом’якшення.

5. Порядок виконання

1. Реалізувати базові математичні функції (нормальний розподіл, логарифми)

2. Створити клас для Гаусівського Bayes

3. Реалізувати підрахунок статистик для кожного класу

4. Додати методи прогнозування з обчисленням ймовірностей

5. Реалізувати Мультиноміальний варіант з підтримкою згладжування

6. Створити функції для векторизації текстів

7. Додати візуалізацію для 2D даних

8. Провести експерименти на різних датасетах

6. Результат

Робоча реалізація різних варіантів Bayes з можливістю навчання на різних типах даних, візуалізацією результатів та порівняльним аналізом з іншими методами.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Класи GaussianNB, MultinomialNB, BernoulliNB

- Методи fit, predict, predict_proba

- Функції для обчислення умовних ймовірностей

- Реалізація різних методів згладжування

- Функції попередньої обробки даних

- Методи візуалізації результатів

2. Продемонструвати роботу додатку:

- Класифікація на синтетичних 2D даних з візуалізацією

- Текстова класифікація з аналізом найважливіших слів

- Порівняння точності різних варіантів алгоритму

- Вплив параметра згладжування на результати

- Аналіз помилкових класифікацій

3. Бути готовим відповісти на питання:

- Як вибір типу Bayes впливає на результати для різних даних?

- Чому логарифмічні обчислення важливі для стабільності?

- Як інтерпретувати ймовірності, що видає модель?

- Які ознаки найбільше впливають на рішення класифікатора?

Тести

1. Основне припущення Bayes класифікатора:

1) Всі ознаки однаково важливі

2) Ознаки умовно незалежні за даного класу

3) Дані мають нормальний розподіл

4) Класи лінійно роздільні

2. Згладжування Лапласа використовується для:

1) Прискорення обчислень

2) Зменшення розмірності даних

3) Уникнення нульових ймовірностей

4) Балансування класів

3. Мультиноміальний Bayes найкраще підходить для:

1) Неперервних ознак

2) Бінарних ознак

3) Частот або лічильників

4) Категоріальних ознак

4. При класифікації знаменник P(X) в теоремі Байєса можна ігнорувати, тому що:

1) Він завжди дорівнює 1

2) Він однаковий для всіх класів

3) Він дуже малий

4) Він не впливає на точність

5. Гаусівський Bayes припускає, що ознаки мають:

1) Рівномірний розподіл

2) Біноміальний розподіл

3) Нормальний розподіл

4) Експоненційний розподіл

Відповіді:

1-2,

2-3,

3-3,

4-2,

5-3

Тести підвищеної складності

1. Чому Bayes часто працює добре для текстової класифікації, незважаючи на очевидну залежність між словами?

1) Тексти насправді мають незалежні слова

2) Для правильної класифікації достатньо правильного порядку класів

3) Алгоритм автоматично враховує залежності

4) Згладжування повністю вирішує проблему

2. При використанні TF-IDF ваг з Bayes:

1) Порушується теоретичне обґрунтування, але часто покращується якість

2) Алгоритм перестає бути ймовірнісним

3) Обчислення стають неможливими

4) Результати завжди погіршуються

3. Complement Bayes відрізняється тим, що:

1) Використовує доповнення множини ознак

2) Моделює ймовірності для всіх класів, крім поточного

3) Працює тільки з бінарною класифікацією

4) Не потребує згладжування

4. Для калібрування ймовірностей Bayes можна використовувати:

1) Ізотонічну регресію або Platt scaling

2) Нормалізацію до [0, 1]

3) Логарифмічне перетворення

4) Збільшення згладжування

5. Semi-supervised Bayes з EM алгоритмом:

1) Використовує тільки розмічені дані

2) Ітеративно оцінює параметри, використовуючи нерозмічені дані

3) Потребує більше розмічених даних

4) Працює гірше звичайного варіанту

6. Feature hashing в Bayes дозволяє:

1) Зменшити час навчання

2) Працювати з необмеженим словником без зберігання всіх слів

3) Покращити точність

4) Автоматично вибрати важливі ознаки

7. Чому Бернуллівський NB може працювати краще Мультиноміального для коротких текстів?

1) Він швидше навчається

2) Відсутність слова також несе інформацію

3) Він використовує менше пам’яті

4) Короткі тексти завжди бінарні

8. Weighted Bayes відрізняється тим, що:

1) Використовує ваги для балансування класів

2) Дозволяє різним ознакам мати різну важливість

3) Автоматично видаляє нерелевантні ознаки

4) Працює тільки з числовими даними

9. При виборі між дискретизацією та Гаусівським NB для числових ознак:

1) Завжди краще дискретизація

2) Залежить від справжнього розподілу даних

3) Гаусівський завжди точніше

4) Різниці немає

10. Lazy learning варіант Bayes:

1) Не зберігає навчальні дані

2) Обчислює статистики локально навколо тестової точки

3) Навчається повільніше

4) Використовує менше пам’яті

Відповіді:

1. 2 - Для ранжування класів за ймовірністю точні значення не критичні

2. 1 - TF-IDF не є справжніми ймовірностями, але емпірично покращує результати

3. 2 - CNB використовує статистики з інших класів для кращої роботи з дисбалансом

4. 1 - Ці методи коригують систематичне зміщення ймовірностей NB

5. 2 - EM дозволяє використовувати інформацію з нерозмічених даних

6. 2 - Хешування дозволяє фіксований розмір ознак незалежно від словника

7. 2 - Для коротких текстів інформація про відсутність слів часто важлива

8. 2 - WNB дозволяє навчити ваги ознак, послаблюючи припущення про рівну важливість

9. 2 - Якщо розподіл далекий від нормального, дискретизація може бути кращою

10. 2 - Локальні статистики можуть краще моделювати локальні залежності

Завдання для самостійної роботи

1. Реалізуйте структурований Bayes, який враховує залежності між групами ознак (наприклад, n-грами в текстах).

2. Створіть адаптивний варіант, який автоматично вибирає тип NB (Gaussian/Multinomial/Bernoulli) на основі характеристик даних.

3. Реалізуйте online Bayes, який може ефективно оновлюватися при надходженні нових даних без повного перенавчання.

4. Розробіть методи візуалізації впливу кожної ознаки на фінальне рішення для конкретного прикладу.

5. Імплементуйте Hierarchical Bayes для класифікації з ієрархічною структурою класів.

6. Створіть ансамбль з різних варіантів Bayes для покращення робастності.

7. Реалізуйте методи автоматичного вибору оптимального параметра згладжування через крос-валідацію.

8. Розробіть Bayes з підтримкою відсутніх значень без необхідності їх заповнення.

9. Імплементуйте kernel Bayes, який може працювати з нелінійними границями рішення.

10. Створіть інтерпретований Bayes, який генерує зрозумілі правила класифікації.

Теми лабораторних робіт

1. Реалізація та порівняння всіх основних варіантів Bayes на різних типах даних.

2. Розробка спам-фільтра з використанням Bayes та аналізом найінформативніших ознак.

3. Створення системи аналізу тональності відгуків з обробкою негацій та модифікаторів.

4. Імплементація medical diagnosis системи з калібруванням ймовірностей для надійних оцінок ризику.

5. Розробка рекомендаційної системи на основі Bayes з холодним стартом.

6. Порівняльний аналіз різних методів згладжування та їх впливу на якість класифікації.

7. Створення multi-label класифікатора документів з ієрархічною структурою категорій.

8. Реалізація real-time класифікатора з обмеженнями на час та пам’ять.

9. Розробка системи виявлення аномалій на основі one-class Bayes.

10. Імплементація privacy-preserving Bayes з диференційною приватністю.

15.4. Нейронні мережі

У цьому розділі ми зануримося в світ нейронних мереж — потужних моделей, натхненних будовою біологічного мозку. Почнемо з найпростішої одиниці обчислення — **перцептрона**, який реалізує лінійне розділення та закладає фундамент для всіх подальших архітектур. Далі розглянемо механізм зворотного поширення помилки (Backpropagation), що дозволяє

ефективно налаштовувати ваги мережі за допомогою градієнтного спуску, забезпечуючи глибоким моделям здатність до складного навчання.

Після ознайомлення з основами навчання одношарових і багатошарових перцептронів перейдемо до спеціалізованих архітектур — згорткових нейронних мереж (CNN), які відзначаються здатністю автоматично витягувати ознаки з двовимірних даних (зображень, спектрограм) за допомогою операцій згортки та підвибірки; а також рекурентних нейронних мереж (RNN), оптимізованих для роботи з послідовностями (текстом, часовими рядами), де важливим є збереження та періодичне оновлення внутрішнього стану.

У наступних підрозділах ми поетапно розглянемо побудову перцептрона й багатошарових мереж, нюанси алгоритму зворотного поширення, базові блоки CNN та RNN, а також наведемо практичні приклади їх застосування. Такий поступ дозволить вам зрозуміти не тільки математичну сутність моделей, а й обрати відповідну архітектуру під конкретні завдання.

Перцептрон Розенблата

Теоретичні відомості

Перцептрон, розроблений Френком Розенблатом у 1957 році, став однією з перших моделей штучних нейронних мереж та заклав основи сучасного машинного навчання. Ця модель була натхненна біологічними нейронами та їх здатністю обробляти інформацію через зважені зв’язки. Перцептрон є найпростішим типом штучної нейронної мережі, що складається з одного нейрона і здатний вирішувати задачі бінарної класифікації для лінійно роздільних даних.

Математична модель перцептрона

Перцептрон моделює процес прийняття рішення як лінійну комбінацію вхідних сигналів з подальшим застосуванням порогової функції активації. Для вектора вхідних ознак $x = (x_1, x_2, \ldots, x_n)$ та вектора ваг $w = (w_1, w_2, \ldots, w_n)$ з порогом b (bias), вихід перцептрона визначається як:

$$y = f\left(\sum_{i=1}^{n} w_i\, x_i + b\right)$$

де f - функція активації. Класичний перцептрон використовує ступінчасту функцію активації:

$$f(z) = \begin{cases} 1, & \text{якщо } z \geq 0 \\ 0, & \text{якщо } z < 0 \end{cases}$$

Для задач класифікації з мітками класів $\{-1, +1\}$ часто використовується функція знаку:

$$f(z) = \text{sign}(z) = \begin{cases} +1, & \text{якщо } z \geq 0 \\ -1, & \text{якщо } z < 0 \end{cases}$$

Геометрична інтерпретація

Перцептрон реалізує лінійний класифікатор, що розділяє простір ознак гіперплощиною. Рівняння розділяючої гіперплощини:

$$w^T x + b = 0$$

Точки по один бік від гіперплощини ($w^T x + b > 0$) класифікуються як клас +1, а по інший бік ($w^T x + b < 0$) - як клас -1. Вектор ваг w визначає нормаль до гіперплощини, а зсув b контролює відстань гіперплощини від початку координат.

Алгоритм навчання перцептрона

Навчання перцептрона базується на ітеративній корекції ваг при неправильній класифікації. Основна ідея полягає в тому, щоб змінювати ваги в напрямку, який зменшує помилку класифікації. Алгоритм працює наступним чином:

1. Ініціалізація: Ваги w та зсув b ініціалізуються малими випадковими значеннями або нулями.

2. Ітеративне оновлення: Для кожного навчального прикладу (x_i, y_i):

- Обчислюється прогноз: $\hat{y}_i = \text{sign}(w^T x_i + b)$

- Якщо прогноз неправильний ($\hat{y}_i \neq y_i$), оновлюються ваги та зсув:

$$w \leftarrow w + \eta \cdot y_i \cdot x_i$$

$$b \leftarrow b + \eta \cdot y_i$$

- Якщо прогноз правильний, ваги залишаються незмінними

3. Збіжність: Процес повторюється до досягнення збіжності (всі приклади класифіковані правильно) або максимальної кількості епох.

Параметр η (eta) називається швидкістю навчання і контролює величину кроку при оновленні ваг. Зазвичай $\eta \in (0,1]$.

Теорема збіжності перцептрона

Ключовою властивістю алгоритму є гарантована збіжність для лінійно роздільних даних.

Теорема Новікова-Розенблата стверджує:

Якщо навчальна вибірка лінійно роздільна з зазором $\gamma > 0$, то алгоритм перцептрона збігається за скінченну кількість кроків. Кількість помилок обмежена величиною:

$$k \leq \frac{R^2}{\gamma^2}$$

де $R = \max_i ||x_i||$ - максимальна норма навчальних прикладів, γ - геометричний зазор між класами.

Розширення базової моделі

Для покращення можливостей перцептрона були розроблені різні модифікації:

1. Перцептрон з margin - модифікація, що намагається знайти розділяючу гіперплощину з максимальним зазором: - Оновлення відбувається не тільки при помилковій класифікації, але й коли приклад потрапляє в зону невизначеності біля границі - Умова оновлення: $y_i(w^T x_i + b) < \delta$, де $\delta > 0$ - параметр зазору

2. Усереднений перцептрон - використовує усереднення всіх векторів ваг протягом навчання: - Зберігає суму всіх векторів ваг: $\bar{w} = \frac{1}{T}\sum_{t=1}^{T} w_t$ - Фінальна модель використовує усереднені ваги, що покращує узагальнення

3. kernel trick - дозволяє працювати з нелінійно роздільними даними через kernel trick:

- Використовує ядрові функції для неявного відображення в простір вищої розмірності

- Дуальне представлення дозволяє ефективно працювати з високорозмірними просторами

Псевдокод алгоритму перцептрона

```
функція НавчитиПерцептрон(X, y, η, макс_епох)
    // X - матриця навчальних прикладів (n × d)
    // y - вектор міток класів {-1, +1}
    // η - швидкість навчання
    // макс_епох - максимальна кількість епох

    n = кількість_прикладів(X)
    d = кількість_ознак(X)

    // Ініціалізація ваг та зсуву
```

```
    w = нулі(d)
    b = 0

    для епоха = 1 до макс_епох виконати
        помилки = 0

        для i = 1 до n виконати
            // Обчислення лінійної комбінації
            z = скалярний_добуток(w, X[i]) + b

            // Прогнозування класу
            якщо z >= 0 тоді
                прогноз = +1
            інакше
                прогноз = -1

            // Перевірка правильності та оновлення
            якщо прогноз != y[i] тоді
                w = w + η * y[i] * X[i]
                b = b + η * y[i]
                помилки = помилки + 1

        // Перевірка збіжності
        якщо помилки == 0 тоді
            вивести "Збіжність досягнута на епосі", епоха
            перервати

    якщо помилки > 0 тоді
        вивести "Не вдалося досягти збіжності за", макс_епох, "епох"

    повернути w, b

функція КласифікуватиПерцептрон(x, w, b)
    // x - вектор ознак для класифікації
    // w - вектор ваг
    // b - зсув

    z = скалярний_добуток(w, x) + b

    якщо z >= 0 тоді
        повернути +1
    інакше
        повернути -1

// Покращена версія з усередненням
```

```
функція НавчитиУсередненийПерцептрон(X, y, η, макс_епох)
    n = кількість_прикладів(X)
    d = кількість_ознак(X)

    w = нулі(d)
    b = 0
    w_сума = нулі(d)
    b_сума = 0
    лічильник = 0

    для епоха = 1 до макс_епох виконати
        для i = 1 до n виконати
            z = скалярний_добуток(w, X[i]) + b
            прогноз = sign(z)

            якщо прогноз != y[i] тоді
                w = w + η * y[i] * X[i]
                b = b + η * y[i]

            // Накопичення для усереднення
            w_сума = w_сума + w
            b_сума = b_сума + b
            лічильник = лічильник + 1

    // Усереднені ваги
    w_усереднені = w_сума / лічильник
    b_усереднений = b_сума / лічильник

    повернути w_усереднені, b_усереднений

// Версія з margin
функція НавчитиПерцептронЗMargin(X, y, η, δ, макс_епох)
    n = кількість_прикладів(X)
    d = кількість_ознак(X)

    w = нулі(d)
    b = 0

    для епоха = 1 до макс_епох виконати
        оновлень = 0

        для i = 1 до n виконати
            margin = y[i] * (скалярний_добуток(w, X[i]) + b)

            // Оновлення якщо margin менший за поріг
```

```
            якщо margin < δ тоді
                w = w + η * y[i] * X[i]
                b = b + η * y[i]
                оновлень = оновлень + 1

        якщо оновлень == 0 тоді
            перервати

    повернути w, b
```

Приклад застосування

Розглянемо детальний приклад навчання перцептрона для класифікації точок на площині.

Навчальні дані:

- Клас +1: (2, 3), (3, 2), (3, 4)

- Клас -1: (1, 1), (0, 2), (1, 0)

Ініціалізація:

- Ваги: $w = (0,0)$

- Зсув: $b = 0$

- Швидкість навчання: $\eta = 1$

Епоха 1:

Приклад 1: (2, 3), y = +1 - $z = 0 \cdot 2 + 0 \cdot 3 + 0 = 0$

- Прогноз: +1 (оскільки $z \geq 0$)

- Правильно, не оновлюємо

Приклад 2: (1, 1), y = -1 - $z = 0 \cdot 1 + 0 \cdot 1 + 0 = 0$

- Прогноз: +1

- Неправильно! Оновлення:

    - $w = (0,0) + 1 \cdot (-1) \cdot (1,1) = (-1,-1)$

    - $b = 0 + 1 \cdot (-1) = -1$

Приклад 3: (3, 2), y = +1 - $z = (-1) \cdot 3 + (-1) \cdot 2 + (-1) = -6$

- Прогноз: -1

- Неправильно! Оновлення:

    - $w = (-1,-1) + 1 \cdot (+1) \cdot (3,2) = (2,1)$

    - $b = -1 + 1 \cdot (+1) = 0$

Процес продовжується…

Після збіжності:

- Фінальні ваги: $w = (1,2)$

- Фінальний зсув: $b = -3$

- Розділяюча пряма: $x_1 + 2x_2 - 3 = 0$

Проблема XOR

Класичний приклад, що демонструє обмеження перцептрона - задача XOR (виключне АБО):

Дані XOR:

- (0, 0) → клас 0

- (0, 1) → клас 1

- (1, 0) → клас 1

- (1, 1) → клас 0

Ці точки не можуть бути розділені однією прямою лінією. Точки одного класу розташовані по діагоналі, що робить неможливим їх лінійне розділення. Це фундаментальне обмеження одношарового перцептрона призвело до значної критики та тимчасової втрати інтересу до нейронних мереж.

Для XOR потрібні дві лінії або нелінійна границя:

- Точки (0,0) та (1,1) належать одному класу

- Точки (0,1) та (1,0) належать іншому класу

- Жодна пряма не може правильно розділити ці класи

Історичний контекст та відродження

Період занепаду (1969-1986)

У 1969 році Марвін Мінський та Сеймур Паперт опублікували книгу “Perceptrons”, де математично довели обмеження одношарових перцептронів:

- Неможливість вирішення XOR та інших нелінійно роздільних задач

- Обмежена виразна здатність для складних функцій

- Відсутність ефективних алгоритмів для багатошарових мереж

Це призвело до “зими штучного інтелекту” - різкого зменшення фінансування та інтересу до нейронних мереж на майже два десятиліття.

Причини відродження (1986-сьогодні)

Алгоритм зворотного поширення помилки (backpropagation):

- Дозволив ефективно навчати багатошарові мережі
- Вирішив проблему XOR через приховані шари
- Забезпечив градієнтну оптимізацію глибоких архітектур

Теоретичні досягнення:

- Універсальна теорема апроксимації: багатошарові мережі можуть апроксимувати будь-яку неперервну функцію
- Розвиток теорії регуляризації та узагальнення
- Зв’язок з kernel methods та SVM

Обчислювальні ресурси:

- Експоненційне зростання обчислювальної потужності
- GPU прискорення для паралельних обчислень
- Розподілені системи для навчання великих моделей

Великі дані:

- Доступність масивних анотованих датасетів
- Інтернет як джерело даних
- Покращення узагальнення з більшою кількістю даних

Архітектурні інновації:

- Згорткові нейронні мережі для комп’ютерного зору
- Рекурентні мережі для послідовних даних
- Attention механізми та трансформери

Сучасне використання перцептрона

Хоча одношаровий перцептрон має обмеження, він залишається важливим з кількох причин:

Навчальний інструмент:

- Простота дозволяє зрозуміти основи нейронних мереж
- Ілюструє ключові концепції: ваги, активація, градієнтний спуск

- База для розуміння складніших архітектур

Компонент складних систем:

- Вихідний шар багатошарових мереж часто є перцептроном
- Лінійні класифікатори в feature space після нелінійних перетворень
- Attention heads в трансформерах використовують лінійні проекції

Ефективність для простих задач:

- Швидке навчання та прогнозування
- Мінімальні вимоги до пам’яті
- Інтерпретованість через аналіз ваг

Online learning:

- Природна адаптація до потокових даних
- Інкрементальне оновлення без перенавчання
- Важливо для edge computing та IoT

Область застосування

Незважаючи на обмеження, перцептрон знаходить застосування в багатьох сферах:

Обробка тексту:

- Бінарна класифікація документів (спам/не спам)
- Визначення тональності на рівні речень
- Фільтрація контенту

Комп’ютерний зір:

- Детекція простих образів
- Попередня обробка зображень
- Компонент каскадних класифікаторів

Фінансовий аналіз:

- Прогнозування тренду (зростання/падіння)
- Оцінка кредитного ризику (схвалити/відхилити)
- Виявлення шахрайства

Медична діагностика:

- Скринінг захворювань (присутнє/відсутнє)

- Аналіз бінарних біомаркерів
- Тріаж пацієнтів

Промислова автоматизація:

- Контроль якості (придатний/брак)
- Детекція аномалій
- Прості системи прийняття рішень

Приклади застосування

Приклад 1: Фільтрація спаму в email

Email провайдер використовує перцептрон для початкової фільтрації: - Ознаки: наявність ключових слів, довжина теми, кількість посилань - Бінарна класифікація: спам/не спам - Швидка обробка мільйонів листів - Точність 85% достатня для першого рівня фільтрації

Приклад 2: Контроль якості на виробництві

Завод використовує перцептрон для виявлення дефектних деталей: - Вхідні дані: розміри, вага, оптичні характеристики - Рішення в реальному часі на конвеєрі - Простота налаштування для нових типів деталей - Інтеграція з PLC системами

Приклад 3: Медичний скринінг

Клініка використовує перцептрон для попереднього скринінгу діабету: - Ознаки: вік, BMI, рівень глюкози, сімейна історія - Швидка оцінка ризику - Направлення на детальне обстеження при позитивному результаті - Зменшення навантаження на лікарів

Переваги та недоліки

Переваги

Простота реалізації та розуміння: - Мінімальний математичний апарат - Легко візуалізувати процес навчання - Прозора інтерпретація результатів

Швидкість навчання та прогнозування: - Лінійна складність відносно кількості ознак - Ефективні оновлення в online режимі - Мінімальні вимоги до пам’яті

Гарантована збіжність: - Теоретичні гарантії для лінійно роздільних даних - Передбачувана поведінка - Відсутність локальних мінімумів

Інтерпретованість: - Ваги показують важливість ознак - Легко зрозуміти причини рішення - Можливість аналізу помилок

Недоліки

Тільки лінійно роздільні задачі: - Неможливість вирішення XOR - Обмежена виразна здатність - Потреба в feature engineering

Чутливість до масштабу ознак: - Великі значення домінують - Необхідність нормалізації - Проблеми з різнорідними даними

Відсутність ймовірнісних оцінок: - Тільки бінарні рішення - Немає оцінки впевненості - Складність калібрування

Проблеми з шумом: - Outliers сильно впливають на границю - Відсутність регуляризації - Можливість overfitting на малих даних

Контрольні запитання

1. Поясніть математичну модель перцептрона та роль кожного компонента у формулі.

2. Чому перцептрон не може вирішити задачу XOR? Наведіть геометричне пояснення.

3. Опишіть алгоритм навчання перцептрона та умови його збіжності.

4. Яка роль швидкості навчання η в алгоритмі? Як її вибір впливає на збіжність?

5. Поясніть теорему збіжності перцептрона та її практичне значення.

6. Які модифікації базового алгоритму покращують узагальнюючу здатність?

7. Чому книга Мінського та Паперта призвела до “зими ШІ”? Які були основні критики?

8. Які технологічні та теоретичні досягнення призвели до відродження інтересу до нейронних мереж?

9. Де сьогодні використовується одношаровий перцептрон, незважаючи на його обмеження?

10. Як перцептрон пов’язаний з сучасними глибокими нейронними мережами?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження перцептрона Розенблата

2. Мета

Вивчити принципи роботи перцептрона, реалізувати алгоритм навчання, дослідити його можливості та обмеження на різних типах даних.

3. Завдання

- Реалізувати базовий алгоритм перцептрона

- Додати візуалізацію процесу навчання

- Продемонструвати проблему XOR

- Реалізувати модифікації: усереднений перцептрон, версію з margin

- Порівняти з іншими лінійними класифікаторами

- Дослідити вплив параметрів на збіжність

4. Опис

Студенти створюють повну реалізацію перцептрона з можливістю візуалізації процесу навчання, аналізу збіжності та демонстрації обмежень на нелінійно роздільних даних.

5. Порядок виконання

1. Реалізувати клас Perceptron з методами fit та predict

2. Створити функції для генерації лінійно роздільних даних

3. Додати візуалізацію границі рішення для 2D даних

4. Реалізувати анімацію процесу навчання

5. Створити датасет XOR та продемонструвати неможливість навчання

6. Реалізувати покращені версії алгоритму

7. Провести експерименти з різними параметрами

8. Порівняти з логістичною регресією та SVM

6. Результат

Працююча реалізація перцептрона з візуалізацією, демонстрацією можливостей та обмежень, аналізом різних модифікацій алгоритму.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас Perceptron з повною реалізацією

- Функції візуалізації границі рішення

- Генератори різних типів даних

- Реалізація модифікованих версій

- Код для експериментів та порівнянь

2. Продемонструвати роботу додатку:

- Навчання на лінійно роздільних даних
- Візуалізація зміни границі під час навчання
- Демонстрація проблеми XOR
- Порівняння різних версій алгоритму
- Аналіз впливу параметрів

3. Бути готовим відповісти на питання:

- Як змінюється кількість ітерацій до збіжності при різних η?
- Чому нормалізація даних важлива для перцептрона?
- Як візуально відрізнити лінійно роздільні дані від нероздільних?
- Які переваги дає усереднення ваг?

Тести

1. Перцептрон може вирішувати задачі:

1) Будь-якої бінарної класифікації
2) Тільки лінійно роздільної класифікації
3) Багатокласової класифікації
4) Нелінійної регресії

2. Функція активації класичного перцептрона:

1) Сигмоїда
2) ReLU
3) Ступінчаста функція
4) Тангенс

3. Алгоритм перцептрона гарантовано збігається:

1) Для будь-яких даних
2) Тільки для лінійно роздільних даних
3) Ніколи не збігається
4) Тільки для нормалізованих даних

4. Проблема XOR демонструє:

1) Повільну збіжність перцептрона

2) Неможливість лінійного розділення

3) Необхідність великих даних

4) Проблеми з ініціалізацією

5. Теорема збіжності перцептрона була доведена:

1) Розенблатом

2) Мінським та Папертом

3) Новіковим

4) Румельхартом

Відповіді:

1-2,

2-3,

3-2,

4-2,

5-3

Тести підвищеної складності

1. Чому перцептрон з нелінійним перетворенням ознак може вирішити XOR?

1) Це неможливо в принципі

2) Перетворення може зробити дані лінійно роздільними в новому просторі

3) Нелінійність змінює алгоритм навчання

4) Це вимагає зміни функції активації

2. Верхня межа кількості помилок перцептрона залежить від:

1) Тільки розміру навчальної вибірки

2) Квадрата відношення максимальної норми до margin

3) Експоненційно від розмірності

4) Швидкості навчання

3. Voted perceptron відрізняється від звичайного тим, що:

1) Використовує голосування кількох випадкових перцептронів

2) Зберігає всі проміжні гіпотези з вагами

3) Навчається на підвибірках даних

4) Використовує іншу функцію втрат

4. Kernel perceptron дозволяє:

1) Прискорити навчання

2) Працювати в просторі високої розмірності без явного обчислення

3) Автоматично вибирати ознаки

4) Уникнути проблеми переобученя

5. Чому pocket algorithm корисний для не повністю роздільних даних?

1) Він знаходить нелінійну границю

2) Зберігає найкращу гіпотезу за всю історію навчання

3) Використовує регуляризацію

4) Змінює функцію активації

6. Зв’язок перцептрона з SVM:

1) Це абсолютно різні алгоритми

2) Перцептрон знаходить будь-яку розділяючу гіперплощину, SVM - оптимальну

3) SVM це узагальнення перцептрона на багато класів

4) Перцептрон це окремий випадок SVM

7. Чому градієнт ступінчастої функції проблематичний для backpropagation?

1) Він занадто великий

2) Він майже скрізь нуль, що блокує оновлення

3) Він від’ємний

4) Він нелінійний

8. Адаптивна швидкість навчання в перцептроні може:

1) Гарантувати збіжність для нероздільних даних

2) Прискорити збіжність та покращити стабільність

3) Вирішити проблему XOR

4) Збільшити виразну здатність

9. Чому перцептрон чутливий до порядку подання прикладів?

1) Це помилкове твердження

2) Online updates залежать від послідовності помилок

3) Алгоритм має пам’ять

4) Функція активації змінюється

10. Margin perceptron намагається:

1) Максимізувати кількість правильно класифікованих прикладів

2) Знайти розділяючу гіперплощину з запасом стійкості

3) Мінімізувати кількість ітерацій

4) Використати нелінійну границю

Відповіді:

1. 2 - Нелінійне перетворення (як $x_3=x_1x_2$) може зробити XOR лінійно роздільним

2. 2 - Теорема Новікова: кількість помилок $\leqslant (R/\gamma)^2$

3. b - Voted perceptron зберігає всі гіпотези з вагами пропорційними часу “виживання”

4. 2 - Kernel trick дозволяє неявно працювати у високорозмірному просторі

5. 2 - Pocket algorithm завжди зберігає версію з найменшою помилкою на навчальній вибірці

6. 2 - Обидва шукають лінійну границю, але з різними критеріями оптимальності

7. 2 - Похідна ступінчастої функції - дельта-функція (0 майже скрізь)

8. 2 - Адаптивне η може пришвидшити збіжність, але не змінює фундаментальні обмеження

9. 2 - Різний порядок призводить до різних послідовностей оновлень і фінальних ваг

10. 2 - Margin perceptron оновлюється навіть для правильно класифікованих прикладів близько до границі

Завдання для самостійної роботи

1. Реалізуйте multiclass перцептрон використовуючи стратегії one-vs-all та one-vs-one.

2. Створіть адаптивний алгоритм, який автоматично підбирає оптимальну швидкість навчання.

3. Реалізуйте kernel perceptron з різними ядровими функціями та порівняйте з SVM.

4. Дослідіть вплив різних методів ініціалізації ваг на швидкість збіжності.

5. Імплементуйте online версію з можливістю донавчання на нових даних.

6. Створіть ансамбль перцептронів з різними початковими умовами.

7. Реалізуйте візуалізацію простору ваг та траєкторії навчання.

8. Розробіть методи регуляризації для перцептрона (L1, L2, dropout).

9. Дослідіть зв'язок між перцептроном та лінійною регресією.

10. Створіть інтерактивну демонстрацію для навчальних цілей.

Теми лабораторних робіт

1. Базова реалізація перцептрона з візуалізацією процесу навчання.

2. Дослідження теореми збіжності: емпірична перевірка верхніх меж.

3. Порівняльний аналіз різних модифікацій: averaged, voted, margin perceptron.

4. Реалізація та аналіз pocket algorithm для зашумлених даних.

5. Kernel perceptron: реалізація та порівняння з лінійною версією.

6. Multiclass extensions: різні стратегії та їх ефективність.

7. Дослідження впливу попередньої обробки даних на продуктивність.

8. Створення каскаду перцептронів для складних задач.

9. Hardware implementation: моделювання перцептрона на FPGA.

10. Історична реконструкція: реалізація оригінального Mark I Perceptron.

Багатошарова нейронна мережа та алгоритм зворотного поширення помилки

Теоретичні відомості

Багатошарова нейронна мережа (Multilayer Neural Network) є розширенням одношарового перцептрона, що дозволяє вирішувати нелінійно роздільні задачі через додавання прихованих шарів нейронів між вхідним та вихідним шарами. Ключовим проривом, що зробив можливим ефективне навчання таких мереж, став алгоритм зворотного поширення помилки (backpropagation), розроблений незалежно кількома дослідниками у 1970-

80х роках та популяризований Румельхартом, Хінтоном та Вільямсом у 1986 році.

Архітектура багатошарової мережі

Багатошарова нейронна мережа прямого поширення (feedforward) складається з кількох шарів нейронів:

Вхідний шар - приймає вхідні дані без обробки. Кількість нейронів відповідає розмірності вхідного вектора ознак.

Приховані шари - один або кілька шарів, що виконують нелінійні перетворення даних. Кожен нейрон прихованого шару отримує зважену суму виходів попереднього шару та застосовує функцію активації.

Вихідний шар - генерує фінальні прогнози мережі. Кількість нейронів залежить від типу задачі (один для бінарної класифікації або регресії, кілька для багатокласової класифікації).

Математично, для мережі з L шарами, активація нейрона j в шарі l обчислюється як:

$$z_j^{(l)} = \sum_{i=1}^{n^{(l-1)}} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)}$$

$$a_j^{(l)} = g^{(l)}(z_j^{(l)})$$

де: - $w_{ji}^{(l)}$ - вага зв'язку від нейрона i в шарі $(l-1)$ до нейрона j в шарі l - $a_i^{(l-1)}$ - активація нейрона i в попередньому шарі - $b_j^{(l)}$ - зсув (bias) нейрона j в шарі l - $g^{(l)}$ - функція активації шару l - $z_j^{(l)}$ - зважена сума входів (pre-activation) - $a_j^{(l)}$ - вихідна активація після застосування функції активації

Функції активації

Нелінійні функції активації є критичними для здатності мережі моделювати складні залежності:

1. Сигмоїда: $\sigma(z) = \frac{1}{1+e^{-z}}$

   - Вихід в діапазоні (0, 1)

   - Гладка та диференційовна

   - Проблема зникаючих градієнтів для великих |z|

2. Гіперболічний тангенс: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

   - Вихід в діапазоні (-1, 1)

- Центрована навколо нуля

- Також страждає від зникаючих градієнтів

3. ReLU (Rectified Linear Unit): $\text{ReLU}(z) = \max(0, z)$

- Проста та ефективна обчислювально

- Не має проблеми зникаючих градієнтів для z > 0

- Проблема “мертвих” нейронів для z < 0

Прямий прохід (Forward Propagation)

Обчислення виходу мережі для заданого входу відбувається послідовно від вхідного до вихідного шару:

1. Вхідні дані подаються на вхідний шар: $a^{(0)} = x$

2. Для кожного наступного шару $l = 1,2,\dots,L$:

- Обчислюється зважена сума: $z^{(l)} = W^{(l)}a^{(l-1)} + b^{(l)}$

- Застосовується функція активації: $a^{(l)} = g^{(l)}(z^{(l)})$

3. Вихід мережі: $\hat{y} = a^{(L)}$

Функція втрат

Для навчання мережі потрібно визначити міру розбіжності між прогнозами та справжніми значеннями:

1. Для регресії - середньоквадратична помилка (MSE):

$$J = \frac{1}{2m}\sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2$$

2. Для бінарної класифікації - бінарна крос-ентропія:

$$J = -\frac{1}{m}\sum_{i=1}^{m}[\, y^{(i)}\log(\hat{y}^{(i)}) + (1 - y^{(i)})\log(1 - \hat{y}^{(i)})]$$

3. Для багатокласової класифікації - категоріальна крос-ентропія:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log(\hat{y}_k^{(i)})$$

де m - кількість прикладів, K - кількість класів.

Алгоритм зворотного поширення помилки

Зворотне поширення є ефективним методом обчислення градієнтів функції втрат відносно всіх параметрів мережі, використовуючи ланцюгове правило диференціювання.

Ключова ідея: помилка на виході розповсюджується назад через мережу, і для кожного шару обчислюється внесок у загальну помилку.

Введемо поняття "помилки" нейрона:

$$\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}}$$

Це показує, наскільки зміна pre-activation $z_j^{(l)}$ впливає на функцію втрат.

Крок 1: Обчислення помилки вихідного шару

Для вихідного шару L з MSE втратами:

$$\delta^{(L)} = (\hat{y} - y) \odot g'^{(L)}(z^{(L)})$$

де $\odot$ означає поелементне множення, g' - похідна функції активації.

Крок 2: Зворотне поширення помилки

Для кожного шару $l = L-1, L-2, \ldots, 2$:

$$\delta^{(l)} = (W^{(l+1)})^T \delta^{(l+1)} \odot g'^{(l)}(z^{(l)})$$

Крок 3: Обчислення градієнтів

Градієнти для ваг та зсувів:

$$\frac{\partial J}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^T$$

$$\frac{\partial J}{\partial b^{(l)}} = \delta^{(l)}$$

Крок 4: Оновлення параметрів

Використовуючи градієнтний спуск:

$$W^{(l)} \leftarrow W^{(l)} - \alpha \frac{\partial J}{\partial W^{(l)}}$$

$$b^{(l)} \leftarrow b^{(l)} - \alpha \frac{\partial J}{\partial b^{(l)}}$$

де α - швидкість навчання.

Математичне обґрунтування

Розглянемо детальне виведення формул для простої мережі з одним прихованим шаром.

Нехай маємо: - Вхід: $x \in \mathbb{R}^{n_0}$ - Прихований шар: $h \in \mathbb{R}^{n_1}$ з вагами $W^{(1)} \in \mathbb{R}^{n_1 \times n_0}$ - Вихід: $y \in \mathbb{R}^{n_2}$ з вагами $W^{(2)} \in \mathbb{R}^{n_2 \times n_1}$

Прямий прохід:

$$z^{(1)} = W^{(1)}x + b^{(1)}$$
$$a^{(1)} = g(z^{(1)})$$
$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$
$$\hat{y} = a^{(2)} = g(z^{(2)})$$

Для MSE втрат $J = \frac{1}{2}||\hat{y} - y||^2$, застосовуємо ланцюгове правило:

$$\frac{\partial J}{\partial W_{ij}^{(2)}} = \frac{\partial J}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_i^{(2)}}\frac{\partial z_i^{(2)}}{\partial W_{ij}^{(2)}}$$

Обчислюючи кожну частину: - $\frac{\partial J}{\partial \hat{y}_i} = \hat{y}_i - y_i$ - $\frac{\partial \hat{y}_i}{\partial z_i^{(2)}} = g'(z_i^{(2)})$ - $\frac{\partial z_i^{(2)}}{\partial W_{ij}^{(2)}} = a_j^{(1)}$

Таким чином:

$$\frac{\partial J}{\partial W_{ij}^{(2)}} = (\hat{y}_i - y_i)g'(z_i^{(2)})a_j^{(1)} = \delta_i^{(2)}a_j^{(1)}$$

Псевдокод алгоритму Backpropagation

```
функція НавчитиНейроннуМережу(X_train, y_train, архітектура, α, епохи
)
    // X_train - матриця навчальних прикладів
    // y_train - вектор/матриця цільових значень
    // архітектура - список кількості нейронів у кожному шарі
    // α - швидкість навчання
    // епохи - кількість проходів по даних

    // Ініціалізація параметрів
    L = довжина(архітектура) - 1  // кількість шарів (без вхідного)
    параметри = ІніціалізуватиПараметри(архітектура)

    для епоха = 1 до епохи виконати
        загальна_втрата = 0

        для кожен (x, y) в (X_train, y_train) виконати
            // Прямий прохід
            активації = ПрямийПрохід(x, параметри)

            // Обчислення втрат
            втрата = ОбчислитиВтрату(активації[L], y)
```

```
            загальна_втрата += втрата

            // Зворотне поширення
            градієнти = ЗворотнєПоширення(активації, y, параметри)

            // Оновлення параметрів
            параметри = ОновитиПараметри(параметри, градієнти, α)

        якщо епоха % 100 == 0 тоді
            вивести "Епоха:", епоха, "Втрата:", загальна_втрата / кіл
ькість_прикладів

    повернути параметри

функція ІніціалізуватиПараметри(архітектура)
    параметри = словник()

    для l = 1 до довжина(архітектура)-1 виконати
        n_in = архітектура[l-1]
        n_out = архітектура[l]

        // Ініціалізація Xavier/He
        якщо функція_активації[l] == "relu" тоді
            scale = sqrt(2.0 / n_in)
        інакше
            scale = sqrt(1.0 / n_in)

        параметри["W" + l] = випадкова_матриця(n_out, n_in) * scale
        параметри["b" + l] = нулі(n_out, 1)

    повернути параметри

функція ПрямийПрохід(x, параметри)
    активації = словник()
    активації[0] = x
    L = кількість_шарів(параметри)

    для l = 1 до L виконати
        z = параметри["W" + l] @ активації[l-1] + параметри["b" + l]
        активації["z" + l] = z

        якщо l < L тоді
            активації[l] = relu(z)  // або інша функція активації
        інакше
            активації[l] = softmax(z)  // для класифікації
```

```
    повернути активації

функція ЗворотнєПоширення(активації, y, параметри)
    градієнти = словник()
    L = кількість_шарів(параметри)
    m = кількість_прикладів

    // Помилка вихідного шару
    dz = активації[L] - y
    градієнти["dW" + L] = (1/m) * dz @ активації[L-1].T
    градієнти["db" + L] = (1/m) * сума(dz, вісь=1, keepdims=True)

    // Зворотне поширення через приховані шари
    для l = L-1 до 1 крок -1 виконати
        da = параметри["W" + (l+1)].T @ dz
        dz = da * relu_похідна(активації["z" + l])

        градієнти["dW" + l] = (1/m) * dz @ активації[l-1].T
        градієнти["db" + l] = (1/m) * сума(dz, вісь=1, keepdims=True)

    повернути градієнти

функція ОновитиПараметри(параметри, градієнти, α)
    L = кількість_шарів(параметри)

    для l = 1 до L виконати
        параметри["W" + l] -= α * градієнти["dW" + l]
        параметри["b" + l] -= α * градієнти["db" + l]

    повернути параметри

// Допоміжні функції
функція relu(z)
    повернути максимум(0, z)

функція relu_похідна(z)
    повернути (z > 0) * 1.0

функція sigmoid(z)
    повернути 1 / (1 + exp(-z))

функція sigmoid_похідна(z)
    s = sigmoid(z)
    повернути s * (1 - s)
```

```
функція softmax(z)
    exp_z = exp(z - макс(z))  // для числової стабільності
    повернути exp_z / сума(exp_z)

// Функції втрат
функція MSE(y_pred, y_true)
    повернути (1/2) * середнє((y_pred - y_true)^2)

функція КросЕнтропія(y_pred, y_true)
    // Для уникнення log(0)
    epsilon = 1e-7
    y_pred = обмежити(y_pred, epsilon, 1 - epsilon)
    повернути -середнє(y_true * log(y_pred))
```

Варіанти та покращення алгоритму

1. Stochastic Gradient Descent (SGD) - оновлення після кожного прикладу:

    - Швидше оновлення, краще для online learning

    - Більш шумний процес навчання

    - Може уникати локальних мінімумів

2. Mini-batch Gradient Descent - компроміс між повним batch та SGD:

    - Використовує підмножини даних (типово 32-256 прикладів)

    - Ефективне використання векторизації

    - Баланс між швидкістю та стабільністю

3. Momentum - накопичення швидкості в напрямку градієнта:

$$v_t = \beta v_{t-1} + (1 - \beta)\nabla J$$

$$\theta_t = \theta_{t-1} - \alpha v_t$$

4. Adam (Adaptive Moment Estimation) - адаптивні швидкості навчання:

```
функція Adam(градієнти, параметри, t, α=0.001, β1=0.9, β2=0.999, ε=1e-8)
    // t - номер ітерації

    якщо не існує m та v тоді
        m = нулі_як(параметри)
        v = нулі_як(параметри)

    // Оновлення biased першого моменту
    m = β1 * m + (1 - β1) * градієнти
```

```
    // Оновлення biased другого моменту
    v = β2 * v + (1 - β2) * градієнти^2

    // Корекція bias
    m_hat = m / (1 - β1^t)
    v_hat = v / (1 - β2^t)

    // Оновлення параметрів
    параметри = параметри - α * m_hat / (sqrt(v_hat) + ε)

    повернути параметри, m, v
```

Приклад застосування

Розглянемо детальний приклад навчання простої мережі для вирішення проблеми XOR.

Архітектура мережі:

- Вхідний шар: 2 нейрони (для x_1, x_2)

- Прихований шар: 2 нейрони з сигмоїдною активацією

- Вихідний шар: 1 нейрон з сигмоїдною активацією

Навчальні дані XOR:

```
X = [[0, 0],    y = [[0],
     [0, 1],         [1],
     [1, 0],         [1],
     [1, 1]]         [0]]
```

Ініціалізація (випадкові значення):

```
W¹ = [[0.5, -0.6],     b¹ = [0.1,
      [0.3,  0.8]]           -0.2]

W² = [[0.9, -0.7]]     b² = [0.3]
```

Епоха 1, приклад 1: x = [0, 0], y = 0

Прямий прохід:

```
z¹ = W¹x + b¹ = [[0.5, -0.6],  * [0,  + [0.1,   = [0.1,
                 [0.3,  0.8]]     0]     -0.2]     -0.2]

a¹ = σ(z¹) = [σ(0.1), σ(-0.2)] = [0.525, 0.450]

z² = W²a¹ + b² = [0.9, -0.7] * [0.525, 0.450] + 0.3 = 0.558

ŷ = σ(z²) = σ(0.558) = 0.636
```

Зворотне поширення:

$$\text{Втрата } J = \tfrac{1}{2}(\hat{y} - y)^2 = \tfrac{1}{2}(0.636 - 0)^2 = 0.202$$

$$\delta^2 = (\hat{y} - y) * \sigma'(z^2) = 0.636 * 0.636 * (1-0.636) = 0.147$$

$$\partial J/\partial W^2 = \delta^2 * a^{1T} = 0.147 * [0.525, 0.450] = [0.077, 0.066]$$
$$\partial J/\partial b^2 = \delta^2 = 0.147$$

$$\delta^1 = W^{2T} * \delta^2 \odot \sigma'(z^1) = [0.9, \quad -0.7] * 0.147 \odot [0.525*(1-0.525), \quad 0.450*(1-0.450)]$$
$$= [0.132, -0.103] \odot [0.249, 0.248] = [0.033, -0.026]$$

$$\partial J/\partial W^1 = \delta^1 * x^T = [0.033, \quad -0.026] * [0, 0] = [[0, 0], \quad [0, 0]]$$
$$\partial J/\partial b^1 = \delta^1 = [0.033, -0.026]$$

Оновлення параметрів (α = 1):

$$W^2 = W^2 - \alpha * \partial J/\partial W^2 = [0.9, -0.7] - [0.077, 0.066] = [0.823, -0.766]$$
$$b^2 = b^2 - \alpha * \partial J/\partial b^2 = 0.3 - 0.147 = 0.153$$

W^1 залишається без змін (градієнт = 0)

$$b^1 = b^1 - \alpha * \partial J/\partial b^1 = [0.1, -0.2] - [0.033, -0.026] = [0.067, -0.174]$$

Процес продовжується для всіх прикладів та епох до досягнення збіжності.

Проблеми класичного backpropagation

Проблема зникаючих градієнтів (Vanishing Gradients):

- При проходженні через багато шарів градієнти множаться на похідні функцій активації

- Для сигмоїди максимальна похідна = 0.25

- Після кількох шарів градієнт стає надто малим для ефективного навчання
- Глибокі мережі (>3-4 шари) практично не навчалися

Проблема вибухаючих градієнтів (Exploding Gradients):

- Протилежна проблема - градієнти експоненційно зростають

- Призводить до нестабільності навчання

- Параметри можуть “вистрибувати” далеко від оптимуму

Повільна збіжність:

- Класичний градієнтний спуск може застрягати в седлових точках

- Різні параметри потребують різних швидкостей навчання

- Важко вибрати оптимальну швидкість навчання

Локальні мінімуми:

- Нейронні мережі мають негладку, неопуклу функцію втрат
- Множина локальних мінімумів та седлових точок
- Градієнтний спуск може застрягти в поганому локальному мінімумі

Після початкового ентузіазму 1980-х, нейронні мережі втратили популярність через:

Обчислювальні обмеження:

- Навчання глибоких мереж вимагало днів або тижнів
- Недостатньо пам’яті для великих мереж
- Відсутність ефективних паралельних обчислень

Теоретичні проблеми:

- Відсутність розуміння, як навчати глибокі мережі
- Проблема зникаючих градієнтів здавалася нездоланною
- SVM та інші методи показували кращі результати

Недостатність даних:

- Глибокі мережі потребують великих обсягів даних
- Анотовані датасети були малими та дорогими
- Overfitting був серйозною проблемою

Відродження (2006-сьогодні)

Кілька ключових інновацій привели до “глибокої революції”:

Попереднє ненаглядове навчання (2006):

- Хінтон показав, як використовувати автоенкодери для ініціалізації
- Дозволило навчати глибокі мережі пошарово
- Вирішило частково проблему зникаючих градієнтів

ReLU активація (2010):

- Проста функція max(0, x) революціонізувала навчання
- Відсутність насичення для позитивних значень
- Прискорення навчання в 5-10 разів

Нормалізація:

- Batch Normalization (2015) стабілізувала навчання

- Layer Normalization для рекурентних мереж
- Дозволило навчати мережі з сотнями шарів

Покращені оптимізатори:

- Adam, RMSprop адаптивно налаштовують швидкість навчання
- Momentum допомагає уникати локальних мінімумів
- Learning rate scheduling покращує збіжність

GPU обчислення:

- Масивна паралелізація матричних операцій
- Прискорення в 10-100 разів
- Зробило можливим експерименти з великими мережами

Регуляризація:

- Dropout (2012) - випадкове вимкнення нейронів

Архітектурні інновації:

- Skip connections (ResNet) - прямі з’єднання через шари
- Attention механізми
- Згорткові та рекурентні архітектури

Сучасний стан

Сьогодні backpropagation залишається основою навчання нейронних мереж, але з численними покращеннями:

Автоматичне диференціювання:

- Фреймворки (TensorFlow, PyTorch) автоматично обчислюють градієнти
- Підтримка складних архітектур
- Динамічні обчислювальні графи

Розподілене навчання:

- Паралелізація по даних та моделі
- Синхронне та асинхронне оновлення
- Навчання моделей з мільярдами параметрів

Спеціалізоване обладнання:

- TPU (Tensor Processing Units)

- Оптимізовані для матричних операцій
- Енергоефективність для мобільних пристроїв

Область застосування

Багатошарові нейронні мережі з backpropagation знайшли застосування практично в усіх сферах:

Комп’ютерний зір:

- Розпізнавання об’єктів та обличь
- Сегментація зображень
- Генерація зображень (GAN)
- Медична діагностика по знімках

Обробка природної мови:

- Машинний переклад
- Чат-боти та діалогові системи
- Аналіз тональності
- Автоматичне резюмування

Розпізнавання мовлення:

- Голосові асистенти
- Транскрипція аудіо
- Синтез мовлення

Рекомендаційні системи:

- Персоналізація контенту
- Прогнозування поведінки користувачів
- Колаборативна фільтрація

Ігри та робототехніка:

- AlphaGo/AlphaZero
- Керування роботами
- Навігація автономних транспортних засобів

Наукові дослідження:

- Прогнозування структури білків
- Моделювання клімату

- Відкриття нових матеріалів

Приклади застосування

Приклад 1: Медична діагностика раку шкіри

Лікарня використовує CNN для аналізу дерматоскопічних зображень: - Архітектура: ResNet-50 з transfer learning

- Навчальні дані: 100,000 анотованих зображень
- Точність: 95% (вище за середнього дерматолога)
- Час аналізу: 0.5 секунди на знімок
- Виявляє меланому на ранніх стадіях

Приклад 2: Автоматичний переклад

Технологічна компанія розробила систему перекладу:

- Архітектура: Transformer з 12 шарами
- Підтримка 100+ мов
- BLEU score: 35+ для основних мовних пар
- Обробка контексту до 512 токенів
- Real-time переклад з затримкою <100мс

Приклад 3: Автономне водіння

Автовиробник використовує ensemble нейронних мереж:

- Детекція об’єктів: YOLO v5
- Семантична сегментація: DeepLab v3
- Прогнозування траєкторій: LSTM мережі
- Sensor fusion: LiDAR + камери
- Прийняття рішень 30 разів на секунду

Переваги та недоліки

Переваги

Універсальна апроксимація:

- Теоретично можуть апроксимувати будь-яку неперервну функцію
- Автоматичне виявлення ознак
- Ієрархічне представлення даних

Масштабованість:

- Продуктивність зростає з кількістю даних
- Паралелізація на GPU/TPU
- Transfer learning для нових задач

Адаптивність:

- Працюють з різними типами даних
- End-to-end навчання
- Continual learning можливості

State-of-the-art результати:

- Перевершують людську продуктивність в багатьох задачах
- Постійне покращення архітектур
- Активна дослідницька спільнота

Недоліки

Потреба в великих даних:

- Мільйони параметрів потребують мільйони прикладів
- Дорога анотація даних
- Проблеми з рідкісними класами

Обчислювальна складність:

- Дні або тижні навчання для великих моделей
- Високе енергоспоживання
- Дорога інфраструктура

Інтерпретованість:

- “Чорна скринька” - важко зрозуміти рішення
- Складність debugging
- Проблеми довіри в критичних застосуваннях

Вразливість:

- Переобучення на навчальних даних

Контрольні запитання

1. Поясніть, чому одношаровий перцептрон не може вирішити XOR, а багатошарова мережа може.

2. Опишіть математично, як обчислюється градієнт для ваг у прихованому шарі використовуючи ланцюгове правило.

3. Що таке проблема зникаючих градієнтів та які методи її вирішення?

4. Поясніть різницю між batch, stochastic та mini-batch gradient descent.

5. Як функція активації ReLU вирішує проблему зникаючих градієнтів?

6. Чому ініціалізація ваг важлива для навчання глибоких мереж?

7. Опишіть роль momentum в оптимізації нейронних мереж.

8. Які основні відмінності між SGD, Adam та RMSprop оптимізаторами?

9. Як Batch Normalization покращує навчання глибоких мереж?

10. Поясніть концепцію transfer learning та її переваги.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація багатошарової нейронної мережі з алгоритмом backpropagation

2. Мета

Глибоко зрозуміти принципи роботи нейронних мереж, реалізувати алгоритм зворотного поширення помилки, дослідити вплив різних параметрів на навчання.

3. Завдання

- Реалізувати багатошарову нейронну мережу “з нуля”

- Реалізувати прямий прохід та backpropagation

- Додати різні функції активації та втрат

- Реалізувати різні оптимізатори

- Візуалізувати процес навчання

- Вирішити проблему XOR та порівняти з перцептроном

4. Опис

Студенти створюють повну реалізацію нейронної мережі без використання високорівневих фреймворків, що дозволяє глибоко зрозуміти всі аспекти алгоритму.

5. Порядок виконання

1. Реалізувати клас NeuralNetwork з гнучкою архітектурою

2. Реалізувати функції активації та їх похідні

3. Створити метод прямого проходу

4. Реалізувати алгоритм backpropagation

5. Додати різні оптимізатори (SGD, Momentum, Adam)

6. Реалізувати функції втрат для різних задач

7. Створити візуалізацію навчання та границь рішення

8. Провести експерименти на різних датасетах

6. Результат

Повнофункціональна реалізація нейронної мережі з можливістю навчання на різних задачах, візуалізацією процесу та аналізом результатів.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас NeuralNetwork з повною реалізацією

- Модулі для різних функцій активації

- Реалізація різних оптимізаторів

- Функції для візуалізації

- Тестові скрипти для різних датасетів

2. Продемонструвати роботу додатку:

- Вирішення проблеми XOR

- Класифікація на складніших 2D датасетах

- Візуалізація зміни функції втрат

- Порівняння різних оптимізаторів

- Вплив архітектури на результати

3. Бути готовим відповісти на питання:

- Як вибір функції активації впливає на навчання?

- Чому нормалізація входів важлива?

- Як діагностувати overfitting/underfitting?

- Які переваги дає mini-batch навчання?

Тести

1. Алгоритм backpropagation базується на:

1) Методі Монте-Карло

2) Ланцюговому правилі диференціювання

3) Лінійній алгебрі

4) Статистичному аналізі

2. Проблема зникаючих градієнтів найбільш виражена для:

1) ReLU активації

2) Лінійної активації

3) Сигмоїдної активації

4) Softmax активації

3. Який оптимізатор використовує адаптивні швидкості навчання?

1) SGD

2) Momentum

3) Adam

4) Gradient Descent

4. Batch Normalization нормалізує:

1) Вхідні дані

2) Ваги мережі

3) Активації в межах міні-батчу

4) Градієнти

5. Skip connections в ResNet вирішують проблему:

1) Перенавчання

2) Зникаючих градієнтів

3) Повільної збіжності

4) Локальних мінімумів

Відповіді:

1-2,

2-3,

3-3,

4-3,

5-2

Тести підвищеної складності

1. Чому Xavier ініціалізація масштабує ваги на 1/√n?

1) Для прискорення навчання

2) Щоб зберегти дисперсію активацій через шари

3) Для уникнення переповнення

4) Це емпіричне правило без обґрунтування

2. Gradient clipping запобігає:

1) Зникаючим градієнтам

2) Вибухаючим градієнтам

3) Перенавчанню

4) Недонавчанню

3. Чому ReLU може призводити до “мертвих” нейронів?

1) Через переповнення при великих значеннях

2) Нульовий градієнт для від’ємних входів назавжди блокує оновлення

3) Занадто швидке навчання

4) Неправильна ініціалізація

4. Learning rate warmup використовується для:

1) Прискорення початкового навчання

2) Стабілізації навчання на початку

3) Зменшення часу навчання

4) Покращення фінальної точності

5. Чому L2 регуляризація еквівалентна weight decay?

1) Вони мають різний математичний зміст

2) L2 додає штраф до втрат, що призводить до зменшення ваг при оновленні

3) Це випадковий збіг

4) Працює тільки для SGD

6. Mixup augmentation працює шляхом:

1) Випадкового обрізання зображень

2) Лінійної інтерполяції між прикладами та їх мітками

3) Додавання шуму до даних

4) Ротації зображень

7. Чому cosine annealing learning rate schedule ефективний?

1) Косинус природньо зменшується

2) Періодичні “розігріви” дозволяють вийти з локальних мінімумів

3) Простота реалізації

4) Теоретичні гарантії збіжності

8. Label smoothing запобігає:

1) Перенавчанню на зашумлених мітках

2) Надмірній впевненості моделі

3) Повільній збіжності

4) Всі відповіді правильні

9. Gradient accumulation дозволяє:

1) Прискорити навчання

2) Емулювати великі батчі на обмеженій пам’яті

3) Покращити точність

4) Зменшити variance градієнтів

10. Connection pruning в нейронних мережах:

1) Видаляє непотрібні зв’язки після навчання

2) Покращує узагальнення

3) Зменшує розмір моделі

4) Всі відповіді правильні

Відповіді:

1. 2 - Xavier підтримує однакову дисперсію через шари для стабільного навчання

2. 3 - Обмеження норми градієнта запобігає експоненційному зростанню

3. 2 - Якщо ReLU отримує від’ємні входи, градієнт нуль і вага не оновлюється

4. b - Поступове збільшення lr дозволяє стабільніше почати навчання

5. 2 - Похідна L2 штрафу призводить до віднімання частини ваги при оновленні

6. 2 - Mixup створює віртуальні приклади між реальними точками даних

7. 2 - Періодичне збільшення lr допомагає exploration простору параметрів

8. 4 - Label smoothing має множинні переваги для робастності та калібрування

9. 2 - Накопичення градієнтів дозволяє використовувати ефективні великі батчі

10. 4 - Pruning одночасно стискає модель та може покращити узагальнення

Завдання для самостійної роботи

1. Реалізуйте варіант backpropagation для рекурентних нейронних мереж (BPTT).

2. Створіть автоматичну систему підбору архітектури мережі для заданої задачі.

3. Реалізуйте second-order методи оптимізації (L-BFGS, природний градієнт).

4. Дослідіть вплив різних методів ініціалізації на швидкість збіжності.

5. Імплементуйте різні види регуляризації та порівняйте їх ефективність.

6. Створіть візуалізацію внутрішніх представлень мережі для розуміння що вона вивчає.

7. Реалізуйте knowledge distillation для стиснення великої мережі в малу.

8. Дослідіть adversarial training для покращення робастності.

9. Імплементуйте quantization для зменшення розміру моделі.

10. Створіть систему continual learning без катастрофічного забування.

Теми лабораторних робіт

1. Базова реалізація MLP та backpropagation з нуля на NumPy.

2. Дослідження впливу архітектури на здатність апроксимації функцій.

3. Порівняльний аналіз різних оптимізаторів на складних ландшафтах.

4. Реалізація та аналіз методів регуляризації (Dropout, BatchNorm, Weight Decay).

5. Візуалізація процесу навчання: градієнти, активації, ваги.

6. Transfer learning: fine-tuning попередньо навченої мережі.

7. Adversarial examples: генерація та захист.

8. Neural Architecture Search: автоматичний пошук оптимальної архітектури.

9. Quantization та pruning для deployment на edge devices.

10. Interpretability: LIME, SHAP, GradCAM для розуміння рішень мережі.

Основи згорткових нейронних мереж (CNN)

Теоретичні відомості

Згорткові нейронні мережі (Convolutional Neural Networks, CNN) є спеціалізованим типом нейронних мереж, розроблених для ефективної обробки даних з сітковою топологією, таких як зображення. Архітектура CNN була натхненна організацією зорової кори тварин та революціонізувала комп'ютерний зір після успіху AlexNet у 2012 році. Основна ідея CNN полягає у використанні операції згортки замість повнозв'язних шарів, що дозволяє виявляти локальні патерни та ієрархічно будувати складні представлення.

Традиційні повнозв'язні мережі мають кілька проблем при роботі з зображеннями:

1. Величезна кількість параметрів - для зображення 224×224×3 (RGB) повнозв'язний шар до 1000 нейронів матиме понад 150 мільйонів параметрів лише в одному шарі.

2. Втрата просторової структури - розгортання зображення в вектор знищує інформацію про сусідство пікселів.

3. Відсутність інваріантності до зсуву - об'єкт у різних частинах зображення вимагає окремого навчання для кожної позиції.

CNN вирішує ці проблеми через три ключові ідеї:

1. Локальна зв'язність - кожен нейрон з'єднаний лише з невеликою областю попереднього шару (рецептивне поле).

2. Спільне використання ваг - один набір ваг (фільтр) застосовується до всього зображення.

3. Просторова ієрархія - прості ознаки комбінуються в складніші через послідовність шарів.

Операція згортки

Математично, дискретна 2D згортка визначається як:

$$(f * g)(i,j) = \sum_m \sum_n f\,(m,n) \cdot g(i-m, j-n)$$

Однак у контексті CNN використовується кореляція (згортка без перевертання ядра):

$$(I * K)(i,j) = \sum_{m=0}^{k_h-1} \sum_{n=0}^{k_w-1} I\,(i+m, j+n) \cdot K(m,n)$$

де: - I - вхідне зображення або карта ознак - K - ядро згортки (фільтр) розміру $k_h \times k_w$ - (i,j) - позиція у вихідній карті ознак

Компоненти CNN

1. Згортковий шар (Convolutional Layer)

Основний будівельний блок CNN. Містить набір фільтрів, що навчаються, кожен з яких виявляє певну ознаку.

Для вхідного тензора розміру $H_{in} \times W_{in} \times D_{in}$ та N фільтрів розміру $F \times F \times D_{in}$, вихід матиме розмір $H_{out} \times W_{out} \times N$, де:

$$H_{out} = \lfloor \frac{H_{in} - F + 2P}{S} \rfloor + 1$$

$$W_{out} = \lfloor \frac{W_{in} - F + 2P}{S} \rfloor + 1$$

де P - padding (доповнення), S - stride (крок).

Padding додає нулі навколо вхідного зображення для контролю розміру виходу: - Valid padding (P = 0): зменшує розмір - Same padding (P = (F-1)/2): зберігає розмір при S = 1 - Full padding: збільшує розмір

Stride визначає крок зсуву фільтра. Більший stride зменшує розмір виходу та збільшує рецептивне поле.

2. Шар субдискретизації (Pooling Layer)

Зменшує просторові розміри представлення, що знижує кількість параметрів та обчислень.

Max Pooling - вибирає максимальне значення з вікна:

$$y_{i,j} = \max_{(m,n) \in R_{i,j}} x_{m,n}$$

Average Pooling - обчислює середнє значення:

$$y_{i,j} = \frac{1}{|R_{i,j}|} \sum_{(m,n) \in R_{i,j}} x_{m,n}$$

де $R_{i,j}$ - область pooling для позиції (i,j).

3. Функції активації

Після згортки зазвичай застосовується нелінійна функція активації:

ReLU: $f(x) = \max(0, x)$ - найпопулярніша через простоту та ефективність

Leaky ReLU: $f(x) = \max(\alpha x, x)$, де α - малий нахил для від'ємних значень

ELU: $f(x) = \begin{cases} x & \text{якщо } x > 0 \\ \alpha(e^x - 1) & \text{якщо } x \leq 0 \end{cases}$

Розглянемо обчислення в згортковому шарі.

Для 3D згортки з кількома каналами:

$$z_{i,j,k}^{(l)} = b_k^{(l)} + \sum_{i'=0}^{f-1} \sum_{j'=0}^{f-1} \sum_{k'=0}^{D_{l-1}-1} w_{i',j',k',k}^{(l)} \cdot a_{si+i',sj+j',k'}^{(l-1)}$$

де: - $z_{i,j,k}^{(l)}$ - pre-activation в позиції (i, j) для k-го фільтра в шарі l - $b_k^{(l)}$ - bias для k-го фільтра - $w_{i',j',k',k}^{(l)}$ - ваги фільтра - $a^{(l-1)}$ - активації попереднього шару - s - stride - f - розмір фільтра

Після застосування функції активації:

$$a_{i,j,k}^{(l)} = g(z_{i,j,k}^{(l)})$$

Зворотне поширення в CNN

Обчислення градієнтів у згорткових шарах вимагає особливого підходу через спільне використання ваг.

Градієнт відносно ваг:

$$\frac{\partial L}{\partial w_{i',j',k',k}^{(l)}} = \sum_{i=0}^{H_{out}-1} \sum_{j=0}^{W_{out}-1} \frac{\partial L}{\partial z_{i,j,k}^{(l)}} \cdot a_{si+i',sj+j',k'}^{(l-1)}$$

Градієнт відносно активацій попереднього шару:

$$\frac{\partial L}{\partial a_{i,j,k'}^{(l-1)}} = \sum_{k=0}^{N_l-1} \sum_{i'} \sum_{j'} \frac{\partial L}{\partial z_{\lfloor (i-i')/s \rfloor, \lfloor (j-j')/s \rfloor, k}^{(l)}} \cdot w_{i',j',k',k}^{(l)}$$

де сума береться тільки для тих (i', j'), які дають цілі індекси.

Градієнт через max pooling:

Max pooling передає градієнт тільки до позиції максимального елемента:

$$\frac{\partial L}{\partial x_{m,n}} = \begin{cases} \frac{\partial L}{\partial y_{i,j}} & \text{якщо } (m,n) = \arg\max_{(p,q) \in R_{i,j}} x_{p,q} \\ 0 & \text{інакше} \end{cases}$$

Рецептивне поле

Рецептивне поле нейрона - це область вхідного зображення, яка впливає на його активацію. Для послідовності згорткових шарів:

$$RF_l = RF_{l-1} + (F_l - 1) \times \prod_{i=1}^{l-1} S_i$$

де RF_l - рецептивне поле в шарі l, F_l - розмір фільтра, S_i - stride в шарі i.

Псевдокод CNN

```
функція CNN_ПрямийПрохід(вхід, параметри)
    // вхід - тензор розміру [batch, height, width, channels]
    // параметри - словник з вагами та конфігурацією шарів

    активація = вхід

    для кожен шар в параметри.шари виконати
        якщо шар.тип == "conv" тоді
            активація = ЗгортковийШар(активація, шар)
        якщо шар.тип == "pool" тоді
            активація = PoolingШар(активація, шар)
        якщо шар.тип == "fc" тоді
            активація = ПовнозвязнийШар(активація, шар)

    повернути активація

функція ЗгортковийШар(вхід, параметри_шару)
    // вхід: [batch, H_in, W_in, D_in]
    // параметри_шару містить: фільтри, bias, stride, padding

    batch, H_in, W_in, D_in = розміри(вхід)
    F = параметри_шару.розмір_фільтра
    S = параметри_шару.stride
    P = параметри_шару.padding
    N = параметри_шару.кількість_фільтрів

    // Додати padding
    якщо P > 0 тоді
        вхід = ДодатиPadding(вхід, P)
        H_in += 2*P
        W_in += 2*P

    // Обчислити розміри виходу
    H_out = (H_in - F) / S + 1
    W_out = (W_in - F) / S + 1
```

```
    вихід = нулі([batch, H_out, W_out, N])

    для b = 0 до batch-1 виконати
        для k = 0 до N-1 виконати
            для i = 0 до H_out-1 виконати
                для j = 0 до W_out-1 виконати
                    // Обчислити згортку для поточної позиції
                    сума = параметри_шару.bias[k]

                    для di = 0 до F-1 виконати
                        для dj = 0 до F-1 виконати
                            для dk = 0 до D_in-1 виконати
                                i_вх = i * S + di
                                j_вх = j * S + dj
                                сума += вхід[b, i_вх, j_вх, dk] *
                                        параметри_шару.фільтри[di, dj,
dk, k]

                    вихід[b, i, j, k] = ReLU(сума)

    повернути вихід

функція MaxPooling(вхід, розмір_вікна, stride)
    batch, H_in, W_in, D = розміри(вхід)
    F = розмір_вікна
    S = stride

    H_out = (H_in - F) / S + 1
    W_out = (W_in - F) / S + 1

    вихід = нулі([batch, H_out, W_out, D])
    маски = нулі([batch, H_out, W_out, D])  // для backprop

    для b = 0 до batch-1 виконати
        для d = 0 до D-1 виконати
            для i = 0 до H_out-1 виконати
                для j = 0 до W_out-1 виконати
                    // Знайти максимум у вікні
                    макс_значення = -нескінченність
                    макс_i = 0
                    макс_j = 0

                    для di = 0 до F-1 виконати
                        для dj = 0 до F-1 виконати
                            i_вх = i * S + di
```

```
                            j_вх = j * S + dj

                            якщо вхід[b, i_вх, j_вх, d] > макс_значен
ня тоді
                                макс_значення = вхід[b, i_вх, j_вх, d
]
                                макс_i = i_вх
                                макс_j = j_вх

                    вихід[b, i, j, d] = макс_значення
                    маски[b, макс_i, макс_j, d] = 1  // для градієнта

    повернути вихід, маски

функція CNN_ЗворотнєПоширення(градієнт_виходу, кеш, параметри)
    // градієнт_виходу - градієнт втрат відносно виходу мережі
    // кеш - збережені активації та маски з прямого проходу

    градієнти = словник()
    поточний_градієнт = градієнт_виходу

    // Проходимо шари у зворотному порядку
    для i = кількість_шарів-1 до 0 крок -1 виконати
        шар = параметри.шари[i]

        якщо шар.тип == "conv" тоді
            поточний_градієнт, град_ваги, град_bias =
                ЗгортковийШар_Назад(поточний_градієнт, кеш[i], шар)
            градієнти["W" + i] = град_ваги
            градієнти["b" + i] = град_bias

        якщо шар.тип == "pool" тоді
            поточний_градієнт = MaxPooling_Назад(
                поточний_градієнт, кеш[i].маски, шар)

    повернути градієнти

функція ЗгортковийШар_Назад(град_вихід, кеш, параметри_шару)
    // град_вихід: [batch, H_out, W_out, N]
    вхід = кеш.вхід  // збережений з прямого проходу

    batch, H_out, W_out, N = розміри(град_вихід)
    F = параметри_шару.розмір_фільтра
    S = параметри_шару.stride
```

```
    // Ініціалізація градієнтів
    град_вхід = нулі_як(вхід)
    град_фільтри = нулі_як(параметри_шару.фільтри)
    град_bias = нулі(N)

    для b = 0 до batch-1 виконати
        для k = 0 до N-1 виконати
            // Градієнт для bias
            град_bias[k] += сума(град_вихід[b, :, :, k])

            для i = 0 до H_out-1 виконати
                для j = 0 до W_out-1 виконати
                    // Градієнт через ReLU
                    якщо кеш.z[b, i, j, k] > 0 тоді
                        град_z = град_вихід[b, i, j, k]
                    інакше
                        град_z = 0

                    // Оновити градієнти фільтрів
                    для di = 0 до F-1 виконати
                        для dj = 0 до F-1 виконати
                            для dk = 0 до D_in-1 виконати
                                i_вх = i * S + di
                                j_вх = j * S + dj

                                град_фільтри[di, dj, dk, k] +=
                                    вхід[b, i_вх, j_вх, dk] * град_z

                                град_вхід[b, i_вх, j_вх, dk] +=
                                    параметри_шару.фільтри[di, dj, dk
, k] * град_z

    // Усереднення градієнтів по батчу
    град_фільтри /= batch
    град_bias /= batch

    повернути град_вхід, град_фільтри, град_bias
```

Приклад застосування

Розглянемо детальний приклад роботи простої CNN для класифікації зображень розміром 6×6 пікселів.

Архітектура мережі:

- Вхід: 6×6×1 (градації сірого)

- Conv1: 2 фільтри 3×3, stride=1, padding=0 → 4×4×2

- MaxPool1: 2×2, stride=2 → 2×2×2

- Flatten → 8 нейронів - FC → 2 класи

Вхідне зображення (спрощене представлення цифри “1”):

```
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
0 0 1 1 0 0
```

Фільтри (ініціалізовані для виявлення вертикальних та горизонтальних ліній):

Фільтр 1 (вертикальні лінії):

```
-1  0  1
-1  0  1
-1  0  1
```

Фільтр 2 (горизонтальні лінії):

```
-1 -1 -1
 0  0  0
 1  1  1
```

Крок 1: Згортка

Для позиції (0,0) з фільтром 1:

```
Вхід[0:3, 0:3]:     Фільтр 1:
0 0 1               -1  0  1
0 0 1         *     -1  0  1
0 0 1               -1  0  1

Результат: 0×(-1) + 0×0 + 1×1 + 0×(-1) + 0×0 + 1×1 + 0×(-1) + 0×0 + 1
×1 = 3
```

Повна карта ознак після фільтра 1:

```
3  3 -3 -3
3  3 -3 -3
3  3 -3 -3
3  3 -3 -3
```

Після ReLU:

```
3 3 0 0
3 3 0 0
3 3 0 0
3 3 0 0
```

Карта ознак після фільтра 2 (виявляє мало горизонтальних ліній):

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

Крок 2: Max Pooling

Для першої карти ознак, вікно 2×2: - Позиція (0,0): max(3,3,3,3) = 3 - Позиція (0,1): max(0,0,0,0) = 0

Результат після pooling:

```
Карта 1:    Карта 2:
3 0         0 0
3 0         0 0
```

Крок 3: Flatten та повнозв'язний шар

Flatten: [3, 0, 3, 0, 0, 0, 0, 0]

Якщо ваги FC налаштовані на виявлення вертикальних ліній: - W для класу "1": [1, 0, 1, 0, 0, 0, 0, 0] - W для класу "0": [0, 1, 0, 1, 1, 1, 1, 1]

Виходи: - Клас "1": 3×1 + 0×0 + 3×1 + … = 6 - Клас "0": 3×0 + 0×1 + 3×0 + … = 0

Після softmax: P("1") ≈ 0.998, P("0") ≈ 0.002

Навчання через backpropagation

При помилковій класифікації градієнти поширюються назад:

1. Градієнт втрат cross-entropy

2. Через FC шар до flatten вектора

3. Reshape до 2×2×2

4. Через max pooling (тільки до позицій максимумів)

5. Через ReLU (нуль для негативних pre-activations)

6. Згортка з перевернутими фільтрами для градієнта входу

7. Накопичення градієнтів для оновлення фільтрів

Область застосування

CNN домінують у задачах комп'ютерного зору та знайшли застосування в багатьох інших областях:

Класифікація зображень:

- Розпізнавання об'єктів (ImageNet, CIFAR)

- Медична діагностика (рентген, МРТ, КТ)

- Розпізнавання облич
- Класифікація документів

Детекція об’єктів:

- YOLO, R-CNN для real-time детекції
- Автономне водіння
- Системи безпеки
- Промислова інспекція

Семантична сегментація:

- U-Net для медичних зображень
- Сегментація відео
- Аналіз супутникових знімків
- Augmented reality

Генерація зображень:

- GAN для створення реалістичних зображень

Обробка відео:

- Розпізнавання дій
- Трекінг об’єктів
- Стабілізація відео
- Компресія відео

Приклади застосування

Приклад 1: Діагностика пневмонії з рентгенівських знімків

Лікарня використовує CNN для автоматичного скринінгу:

- Архітектура: DenseNet-121 з transfer learning
- Датасет: 100,000 анотованих рентгенограм грудної клітки
- Preprocessing: нормалізація, augmentation (ротація, зум)

Приклад 2: Система контролю якості на виробництві

Завод електроніки використовує CNN для виявлення дефектів:

- Архітектура: Custom CNN з 6 згортковими шарами
- Вхід: 512×512 RGB зображення плат
- Класи дефектів: подряпини, відсутні компоненти, погані паяння –

Data collection: 50,000 зображень з 8 камер

- Особливості:
    - Attention механізм для фокусування на проблемних областях
    - Multi-scale обробка для різних розмірів дефектів

Приклад 3: Розпізнавання емоцій для маркетингових досліджень

Компанія аналізує реакції на рекламу:

- Модель: MobileNet для edge computing
- Виявлення облич: MTCNN
- Класифікація емоцій: 7 базових емоцій
- Обробка: - Real-time відео з веб-камер

Переваги та недоліки

Переваги

Автоматичне виявлення ознак:

- Не потребує ручного налаштування
- Навчається ієрархії ознак від простих до складних
- Адаптується до специфіки даних

Інваріантність до зсуву

Ефективність параметрів:

- Значно менше параметрів ніж повнозв’язні мережі
- Локальна зв’язність відповідає структурі зображень
- Можливість дуже глибоких архітектур

Паралелізація:

- Згортка природно паралелізується
- Ефективне використання GPU
- Пакетна обробка

Недоліки

Потреба в великих даних:

- Мільйони параметрів вимагають багато прикладів
- Data augmentation частково вирішує
- Дорога анотація для специфічних доменів

Обчислювальна складність:

- Високі вимоги до GPU для навчання
- Енергоспоживання для великих моделей
- Складність deployment на edge пристроях

Чутливість до adversarial прикладів:

- Малі збурення можуть обдурити мережу
- Проблеми безпеки в критичних застосуваннях
- Потреба в робастному навчанні

Обмежена інтерпретованість:

- Важко зрозуміти причини рішень
- Візуалізація фільтрів дає обмежене розуміння
- Проблеми довіри в медицині та фінансах

Фіксований розмір входу:

- Більшість архітектур вимагають фіксований розмір
- Втрата інформації при масштабуванні
- Spatial pyramid pooling частково вирішує

Труднощі з малими об’єктами:

- Pooling втрачає дрібні деталі
- Потреба в multi-scale підходах

Контрольні запитання

1. Поясніть, чому операція згортки ефективніша за повнозв’язні шари для обробки зображень.

2. Як padding та stride впливають на розмір вихідної карти ознак? Наведіть формули.

3. Яка роль max pooling в CNN та які є альтернативи?

4. Поясніть концепцію рецептивного поля та як воно змінюється через шари.

5. Чому ReLU стала стандартною функцією активації в CNN?

6. Як обчислюються градієнти для спільних ваг у згортковому шарі?

7. Опишіть основні відмінності між LeNet, AlexNet та ResNet.

8. Що таке dilated (atrous) convolution та коли вона корисна?

9. Як transfer learning працює з CNN та чому він ефективний?

10. Поясніть проблему vanishing gradients в глибоких CNN та методи її вирішення.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження згорткової нейронної мережі

2. Мета

Зрозуміти принципи роботи CNN, реалізувати основні компоненти, навчитися проектувати архітектури для різних задач.

3. Завдання

- Реалізувати операцію згортки та max pooling

- Створити просту CNN архітектуру

- Реалізувати backpropagation для CNN

- Візуалізувати навчені фільтри та активації

- Порівняти з повнозв'язною мережею

- Експериментувати з різними архітектурами

4. Опис

Студенти створюють власну реалізацію CNN, що дозволяє глибоко зрозуміти всі аспекти роботи згорткових мереж.

5. Порядок виконання

1. Реалізувати функцію 2D згортки з підтримкою padding та stride

2. Реалізувати max pooling та average pooling

3. Створити клас ConvLayer та PoolingLayer

4. Реалізувати backward pass для кожного типу шару

5. Зібрати просту CNN (Conv-Pool-Conv-Pool-FC)

6. Навчити на MNIST або CIFAR-10

7. Візуалізувати фільтри першого шару

8. Порівняти точність та швидкість з MLP

6. Результат

Робоча реалізація CNN з можливістю навчання на реальних датасетах, візуалізацією внутрішніх представлень та аналізом продуктивності.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Реалізація операції згортки
- Класи різних типів шарів
- Backward propagation для CNN
- Функції візуалізації
- Скрипти навчання та тестування

2. Продемонструвати роботу додатку:

- Навчання на MNIST/CIFAR-10
- Візуалізація фільтрів та feature maps
- Графіки навчання (loss, accuracy)
- Порівняння різних архітектур
- Confusion matrix на тестовій вибірці

3. Бути готовим відповісти на питання:

- Як вибір розміру фільтра впливає на результати?
- Чому глибші мережі не завжди кращі?
- Як інтерпретувати візуалізовані фільтри?
- Які обмеження має реалізована архітектура?

Тести

1. Для входу 32×32×3 та 64 фільтрів 5×5 зі stride=1, padding=2, розмір виходу:

1) 32×32×64
2) 28×28×64
3) 30×30×64
4) 16×16×64

2. Max pooling з вікном 2×2 та stride=2:

1) Збільшує розмір в 2 рази
2) Зменшує розмір в 2 рази
3) Не змінює розмір

4) Зменшує розмір в 4 рази

3. Основна перевага CNN над MLP для зображень:

1) Швидше навчання

2) Менше параметрів та кращі індуктивні упередження

3) Простіша реалізація

4) Не потребує GPU

4. ReLU в CNN вирішує проблему:

1) Перенавчання

2) Зникаючих градієнтів

3) Великої кількості параметрів

4) Повільної збіжності

5. Dilated convolution використовується для:

1) Зменшення параметрів

2) Збільшення рецептивного поля без втрати роздільності

3) Прискорення обчислень

4) Покращення точності

Відповіді:

1-1,

2-2,

3-2,

4-2,

5-2

Тести підвищеної складності

1. Чому пакетнамнормалізація в CNN зазвичай застосовується після згортки, але перед активацією?

1) Для прискорення обчислень

2) Нормалізація pre-activations стабілізує навчання

3) Це не має значення

4) Для зменшення параметрів

2. Depthwise separable convolution розділяє стандартну згортку на:

1) Дві послідовні згортки

2) Spatial та channel-wise операції

3) Forward та backward passes

4) Real та imaginary частини

3. У ResNet skip connections додаються:

1) Після кожної згортки

2) Через кожні 2-3 шари

3) Тільки в кінці мережі

4) Випадково під час навчання

4. Global average pooling замість FC шарів:

1) Збільшує кількість параметрів

2) Зменшує overfitting та кількість параметрів

3) Погіршує точність

4) Прискорює навчання

5. Transpose convolution (deconvolution) використовується для:

1) Інвертування згортки

2) Збільшення просторових розмірів

3) Зменшення параметрів

4) Покращення градієнтів

6. У який спосіб 1×1 згортка може бути корисною?

1) Для зміни кількості каналів

2) Для додавання нелінійності

3) Для зменшення обчислень перед більшими згортками

4) Всі відповіді правильні

7. Чому grouped convolution ефективна?

1) Зменшує кількість параметрів та обчислень

2) Покращує точність

3) Прискорює backward pass

4) Спрощує реалізацію

8. Просторовий пірамідальний pooling дозволяє:

1) Прискорити pooling

2) Працювати з входами різного розміру

3) Покращити градієнти

4) Зменшити пам’ять

9. Channel attention (як в SENet) працює через:

1) Збільшення кількості каналів

2) Адаптивне зважування важливості каналів

3) Видалення неважливих каналів

4) Перемішування каналів

10. MixUp augmentation в контексті CNN:

1) Змішує пікселі одного зображення

2) Лінійно інтерполює між зображеннями та мітками

3) Додає шум до ваг

4) Перемішує канали

Відповіді:

1. 2 - BN нормалізує розподіл до активації для кращої стабільності

2. 2 - Розділення на depthwise (spatial) та pointwise (1×1) згортки

3. 2 - Skip connections через residual blocks для кращого градієнтного потоку

4. 2 - GAP має регуляризуючий ефект та видаляє більшість параметрів

5. 2 - Upsampling через навчені фільтри для збільшення роздільності

6. 4 - 1×1 згортки мають багато застосувань включаючи bottlenecks

7. 1 - Розділення каналів на групи значно зменшує обчислення

8. 2 - SPP створює фіксований вихід незалежно від розміру входу

9. 2 - SE блоки навчаються важливості кожного каналу

10. 2 - MixUp створює нові приклади через опуклі комбінації

Завдання для самостійної роботи

1. Реалізуйте різні види padding (reflect, replicate) та дослідіть їх вплив на граничні артефакти.

2. Створіть візуалізацію рецептивних полів для різних архітектур CNN.

3. Імплементуйте dilated convolution та порівняйте з звичайною для задач сегментації.

4. Реалізуйте depthwise separable convolutions та порівняйте ефективність з стандартними.

5. Дослідіть вплив різних функцій активації (ReLU, ELU, Swish) на навчання CNN.

6. Створіть інструмент для візуалізації того, які частини зображення активують певні фільтри.

7. Реалізуйте spatial transformer networks для навчання інваріантності до трансформацій.

8. Порівняйте різні стратегії ініціалізації ваг для CNN.

9. Імплементуйте attention механізм для CNN та оцініть покращення.

10. Створіть автоматичний пошук архітектури CNN для заданого датасету.

Теми лабораторних робіт

1. Базова реалізація CNN та порівняння з MLP на MNIST.

2. Дослідження впливу архітектурних рішень на продуктивність CNN.

3. Transfer learning: fine-tuning попередньо навчених моделей.

4. Візуалізація та інтерпретація навчених CNN.

5. Реалізація сучасних архітектур: ResNet, DenseNet.

6. Data augmentation стратегії для покращення узагальнення.

7. Adversarial attacks та захист CNN.

8. Оптимізація CNN для мобільних пристроїв.

9. Semantic segmentation з використанням fully convolutional networks.

10. Object detection: реалізація спрощеного YOLO.

Основи рекурентних нейронних мереж (RNN)

Теоретичні відомості

Рекурентні нейронні мережі (Recurrent Neural Networks, RNN) є класом нейронних мереж, спеціально розроблених для обробки послідовних даних змінної довжини. На відміну від традиційних нейронних мереж прямого поширення, RNN мають циклічні зв’язки, що дозволяють їм зберігати інформацію про попередні елементи послідовності. Ця властивість робить

RNN ідеальними для задач, де контекст та порядок даних мають критичне значення, таких як обробка природної мови, аналіз часових рядів та розпізнавання мовлення.

Основна концепція RNN полягає в тому, що мережа обробляє вхідну послідовність крок за кроком, підтримуючи прихований стан (hidden state), який служить "пам'яттю" мережі. На кожному кроці часу t мережа отримує поточний вхід x_t та прихований стан з попереднього кроку h_{t-1}, і обчислює новий прихований стан h_t та вихід y_t.

Математично це можна записати як:

$$h_t = f(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$
$$y_t = g(W_{hy}h_t + b_y)$$

де: - h_t - прихований стан у момент часу t - x_t - вхід у момент часу t - y_t - вихід у момент часу t - W_{hh} - матриця ваг для рекурентних зв'язків - W_{xh} - матриця ваг для вхідних зв'язків - W_{hy} - матриця ваг для вихідних зв'язків - b_h, b_y - вектори зсувів - f, g - функції активації (зазвичай tanh або ReLU для f, та softmax або лінійна для g)

Розгортання в часі

Для розуміння та навчання RNN використовується концепція розгортання в часі (unfolding in time). Циклічна структура RNN може бути представлена як послідовність копій мережі, по одній для кожного кроку часу, де кожна копія передає свій прихований стан наступній.

Для послідовності довжиною T, розгорнута мережа матиме T копій базової RNN комірки, з'єднаних послідовно. Це перетворює рекурентну архітектуру в глибоку мережу прямого поширення, що дозволяє застосувати стандартний алгоритм backpropagation.

Типи RNN архітектур

1. One-to-Many: Один вхід породжує послідовність виходів. Використовується для генерації тексту або музики.

2. Many-to-One: Послідовність входів дає один вихід. Застосовується для класифікації послідовностей або аналізу настроїв.

3. Many-to-Many (синхронний): Кожен елемент вхідної послідовності має відповідний вихід. Використовується для міток послідовностей.

4. Many-to-Many (асинхронний): Спочатку обробляється вся вхідна послідовність, потім генерується вихідна. Застосовується для машинного перекладу.

Зворотне поширення в часі (BPTT)

Навчання RNN відбувається через алгоритм зворотного поширення в часі (BPTT), який є розширенням стандартного backpropagation для розгорнутої в часі мережі.

Для функції втрат $L = \sum_{t=1}^{T} L_t$, де L_t - втрата на кроці t, градієнти обчислюються наступним чином:

Градієнт по вихідних вагах:

$$\frac{\partial L}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial y_t} \frac{\partial y_t}{\partial W_{hy}}$$

Градієнт по прихованому стану:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L_t}{\partial h_t} + \frac{\partial L}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_t}$$

Це рекурсивне співвідношення показує, що градієнт по h_t залежить від градієнтів всіх наступних кроків часу.

Градієнт по рекурентних вагах:

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}$$

Обчислення $\frac{\partial h_t}{\partial h_k}$ включає добуток багатьох якобіанів:

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} W_{hh}^T \operatorname{diag}(f'(W_{hh} h_{i-1} + W_{xh} x_i + b_h))$$

Проблеми класичних RNN

1. Зникаючі градієнти: При множенні багатьох малих значень (похідних функції активації) градієнти експоненційно зменшуються, що робить навчання довгих залежностей практично неможливим.

2. Вибухаючі градієнти: У деяких випадках градієнти можуть експоненційно зростати, призводячи до нестабільності навчання.

3. Короткострокова пам’ять: Через зникаючі градієнти, класичні RNN ефективно зберігають інформацію лише про кілька останніх кроків.

Методи вирішення проблем

1. Gradient Clipping: Обмеження норми градієнта для запобігання вибуху:

$$\text{якщо } ||g|| > \text{threshold, то } g = \frac{\text{threshold}}{||g||} \cdot g$$

2. Ініціалізація ваг: Використання ортогональної або ідентичної ініціалізації для рекурентних ваг.

3. Покращені архітектури: LSTM та GRU, які використовують гейти для контролю потоку інформації.

Варіанти функцій активації

1. Tanh: Класичний вибір для RNN

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Вихід в діапазоні [-1, 1], центрований навколо нуля.

2. ReLU: Може допомогти з зникаючими градієнтами

$$\text{ReLU}(x) = \max(0, x)$$

Але може призвести до вибухаючих активацій.

3. LSTM-style activations: Комбінація sigmoid та tanh для різних компонентів.

Псевдокод базової RNN

```
функція RNN_ПрямийПрохід(X, W_xh, W_hh, W_hy, b_h, b_y, h_0)
    // X - вхідна послідовність розміру [T, input_size]
    // h_0 - початковий прихований стан
    // Повертає: Y - виходи, H - всі приховані стани

    T = довжина(X)
    hidden_size = розмір(h_0)

    H = новий_масив(T+1, hidden_size)
    Y = новий_масив(T, output_size)

    H[0] = h_0

    для t = 1 до T виконати
        // Обчислення прихованого стану
        z_h = W_xh @ X[t] + W_hh @ H[t-1] + b_h
        H[t] = tanh(z_h)
```

```
        // Обчислення виходу
        z_y = W_hy @ H[t] + b_y
        Y[t] = softmax(z_y)  // або інша функція залежно від задачі

    повернути Y, H

функція RNN_BPTT(X, Y_true, W_xh, W_hh, W_hy, b_h, b_y, h_0)
    // Обчислює градієнти через BPTT
    // Y_true - справжні мітки

    // Прямий прохід
    Y_pred, H = RNN_ПрямийПрохід(X, W_xh, W_hh, W_hy, b_h, b_y, h_0)

    T = довжина(X)

    // Ініціалізація градієнтів
    dW_xh = нулі_як(W_xh)
    dW_hh = нулі_як(W_hh)
    dW_hy = нулі_як(W_hy)
    db_h = нулі_як(b_h)
    db_y = нулі_як(b_y)

    dh_next = нулі_як(h_0)

    // Зворотний прохід через час
    для t = T до 1 крок -1 виконати
        // Градієнт втрат по виходу
        dy = Y_pred[t] - Y_true[t]  // для cross-entropy з softmax

        // Градієнти для вихідного шару
        dW_hy += dy @ H[t].T
        db_y += dy

        // Градієнт по прихованому стану
        dh = W_hy.T @ dy + dh_next

        // Градієнт через tanh
        dh_raw = dh * (1 - H[t]^2)  // похідна tanh

        // Градієнти для прихованого шару
        dW_xh += dh_raw @ X[t].T
        dW_hh += dh_raw @ H[t-1].T
        db_h += dh_raw

        // Градієнт для попереднього кроку
```

```
        dh_next = W_hh.T @ dh_raw

    // Gradient clipping
    градієнти = [dW_xh, dW_hh, dW_hy, db_h, db_y]
    градієнти = GradientClipping(градієнти, max_norm=5.0)

    повернути градієнти

функція GradientClipping(градієнти, max_norm)
    total_norm = 0

    для grad в градієнти виконати
        total_norm += сума(grad^2)

    total_norm = sqrt(total_norm)

    якщо total_norm > max_norm тоді
        для i = 0 до довжина(градієнти)-1 виконати
            градієнти[i] = градієнти[i] * max_norm / total_norm

    повернути градієнти

функція НавчитиRNN(X_train, Y_train, epochs, learning_rate)
    // Ініціалізація параметрів
    W_xh = випадкова_матриця(hidden_size, input_size) * 0.01
    W_hh = ортогональна_матриця(hidden_size, hidden_size)
    W_hy = випадкова_матриця(output_size, hidden_size) * 0.01
    b_h = нулі(hidden_size)
    b_y = нулі(output_size)

    для epoch = 1 до epochs виконати
        total_loss = 0

        для batch в розбити_на_батчі(X_train, Y_train) виконати
            h_0 = нулі(hidden_size)

            // Обчислення градієнтів
            градієнти = RNN_BPTT(batch.X, batch.Y, W_xh, W_hh, W_hy,
b_h, b_y, h_0)

            // Оновлення параметрів
            W_xh -= learning_rate * градієнти.dW_xh
            W_hh -= learning_rate * градієнти.dW_hh
            W_hy -= learning_rate * градієнти.dW_hy
            b_h -= learning_rate * градієнти.db_h
```

```
            b_y -= learning_rate * градієнти.db_y

            // Обчислення втрат
            Y_pred, _ = RNN_ПрямийПрохід(batch.X, W_xh, W_hh, W_hy, b
_h, b_y, h_0)
            batch_loss = CrossEntropy(Y_pred, batch.Y)
            total_loss += batch_loss

        вивести "Epoch:", epoch, "Loss:", total_loss / кількість_батч
ів

    повернути W_xh, W_hh, W_hy, b_h, b_y

// Функція для генерації тексту
функція ГенеруватиПослідовність(seed, W_xh, W_hh, W_hy, b_h, b_y, len
gth)
    h = нулі(hidden_size)
    результат = [seed]
    x = OneHotEncode(seed)

    для i = 1 до length виконати
        // Обчислення наступного стану
        h = tanh(W_xh @ x + W_hh @ h + b_h)

        // Обчислення ймовірностей
        y = softmax(W_hy @ h + b_y)

        // Вибірка наступного символу
        next_idx = ВибіркаЗРозподілу(y)
        результат.додати(next_idx)

        // Підготовка наступного входу
        x = OneHotEncode(next_idx)

    повернути результат
```

Дванаправлені RNN

Двонаправлені RNN обробляють послідовність в обох напрямках, що дозволяє використовувати контекст як з минулого, так і з майбутнього:

$$\overrightarrow{h_t} = f(W_{x\overrightarrow{h}}x_t + W_{\overrightarrow{hh}}\overrightarrow{h_{t-1}} + b_{\overrightarrow{h}})$$
$$\overleftarrow{h_t} = f(W_{x\overleftarrow{h}}x_t + W_{\overleftarrow{hh}}\overleftarrow{h_{t+1}} + b_{\overleftarrow{h}})$$
$$y_t = g(W_{\overrightarrow{hy}}\overrightarrow{h_t} + W_{\overleftarrow{hy}}\overleftarrow{h_t} + b_y)$$

Глибокі RNN

Глибокі RNN мають кілька шарів рекурентних комірок:

$$h_t^{(l)} = f(W_{hh}^{(l)} h_{t-1}^{(l)} + W_{hx}^{(l)} h_t^{(l-1)} + b_h^{(l)})$$

де l - номер шару, $h_t^{(0)} = x_t$.

Приклад застосування

Розглянемо детальний приклад використання RNN для простої задачі передбачення наступного символу в послідовності.

Задача: Навчити RNN передбачати наступну літеру в слові “HELLO”.

Підготовка даних:

- Алфавіт: {H, E, L, O}

- Кодування: H→0, E→1, L→2, O→3

- One-hot vectors: H→[1,0,0,0], E→[0,1,0,0], L→[0,0,1,0], O→[0,0,0,1]

Навчальні пари: - H → E - E → L - L → L - L → O

Архітектура:

- Вхідний розмір: 4 (one-hot)

- Прихований розмір: 3

- Вихідний розмір: 4 (ймовірності для кожної літери)

Ініціалізація (спрощені значення):

```
W_xh = [[0.1, 0.2, -0.1, 0.0],    W_hh = [[0.5, -0.3, 0.2],
        [0.3, -0.2, 0.1, 0.2],            [0.1, 0.4, -0.2],
        [-0.1, 0.1, 0.3, -0.2]]           [0.2, -0.1, 0.3]]

W_hy = [[0.2, -0.1, 0.3],
        [0.1, 0.3, -0.2],
        [-0.2, 0.2, 0.1],
        [0.3, -0.1, 0.2]]
```

Крок 1: Вхід ‘H’ [1,0,0,0]

Початковий стан: h_0 = [0, 0, 0]

```
z_h = W_xh @ [1,0,0,0] + W_hh @ [0,0,0] + b_h
    = [0.1, 0.3, -0.1] + [0, 0, 0] + [0, 0, 0]
    = [0.1, 0.3, -0.1]

h₁ = tanh([0.1, 0.3, -0.1]) ≈ [0.10, 0.29, -0.10]

y₁ = softmax(W_hy @ h₁ + b_y)
```

```
   = softmax([0.05, 0.07, -0.01, 0.04])
   ≈ [0.25, 0.26, 0.24, 0.25]
```
Цільовий вихід: E [0,1,0,0]

Втрата: CrossEntropy ≈ -log(0.26) ≈ 1.35

Крок 2: Вхід ‘E’ [0,1,0,0]

```
z_h = W_xh @ [0,1,0,0] + W_hh @ [0.10, 0.29, -0.10]
h₂ = tanh(z_h) ≈ [0.15, 0.35, 0.05]
```
Процес продовжується для всіх символів…

Зворотне поширення:

Для останнього кроку (L→O):

```
dy = y_pred - y_true = [0.25, 0.24, 0.26, 0.25] - [0, 0, 0, 1]
                     = [0.25, 0.24, 0.26, -0.75]

dW_hy += dy @ h₄.T
dh = W_hy.T @ dy
```
Градієнти поширюються назад через всі кроки часу.

Результати навчання

Після достатньої кількості епох, мережа навчається правильним переходам: - P(E|H) ≈ 0.9 - P(L|E) ≈ 0.9 - P(L|L) ≈ 0.5, P(O|L) ≈ 0.5 - P(end|O) ≈ 0.9

Область застосування

RNN знайшли широке застосування в задачах, пов’язаних з послідовними даними:

Обробка природної мови:

- Моделювання мови та генерація тексту

- Машинний переклад

- Аналіз тональності

- Розпізнавання іменованих сутностей

- Відповіді на питання

Розпізнавання та синтез мовлення:

- Перетворення мовлення в текст

- Синтез мовлення з тексту

- Розпізнавання мовця

- Виявлення емоцій у мовленні

Аналіз часових рядів:

- Прогнозування фінансових ринків
- Передбачення погоди
- Аналіз медичних сигналів (ЕКГ, ЕЕГ)
- Прогнозування енергоспоживання

Музика та аудіо:

- Генерація музики
- Класифікація жанрів
- Транскрипція музики
- Розділення джерел звуку

Відео аналіз:

- Розпізнавання дій
- Субтитрування відео
- Прогнозування наступних кадрів
- Відстеження об’єктів

Біоінформатика:

- Аналіз послідовностей ДНК/РНК
- Передбачення структури білків
- Класифікація генів

Переваги та недоліки

Переваги

Обробка змінної довжини:

- Природна робота з послідовностями будь-якої довжини
- Не потребує padding або truncation
- Ефективне використання пам’яті

Збереження контексту:

- Прихований стан зберігає інформацію про всю попередню послідовність
- Можливість моделювання складних залежностей
- Підходить для контекстно-залежних задач

Спільні параметри:

- Одні й ті ж ваги використовуються на всіх кроках
- Менше параметрів порівняно з повнозв’язними мережами
- Кращі узагальнюючі властивості

Гнучкість архітектури:

- Різні конфігурації вхід-вихід
- Можливість bidirectional обробки
- Легко комбінується з іншими архітектурами

Online обробка:

- Можливість обробки потокових даних
- Прогнози в реальному часі
- Інкрементальне навчання

Недоліки

Проблема довгострокових залежностей:

- Зникаючі/вибухаючі градієнти
- Важко навчити залежності на великих відстанях
- Потреба в спеціальних архітектурах (LSTM/GRU)

Послідовна природа:

- Неможливість паралелізації в часі
- Повільне навчання на довгих послідовностях
- Обмежена швидкість inference

Складність налаштування:

- Багато гіперпараметрів
- Чутливість до ініціалізації
- Необхідність gradient clipping

Обмежена пам’ять:

- Фіксований розмір прихованого стану
- Інформація може “забуватися”
- Bottleneck для довгих послідовностей

Інтерпретованість:

- Важко зрозуміти, що саме зберігається в прихованому стані

- Складність debug

- Проблеми з поясненням рішень

Контрольні запитання

1. Поясніть основну відмінність між RNN та традиційними нейронними мережами прямого поширення.

2. Що таке прихований стан в RNN та яку роль він відіграє в обробці послідовностей?

3. Опишіть процес розгортання RNN в часі та його значення для навчання.

4. Як працює алгоритм BPTT та чим він відрізняється від звичайного backpropagation?

5. Поясніть проблему зникаючих градієнтів в RNN та її наслідки.

6. Які існують типи RNN архітектур за співвідношенням вхід-вихід?

7. Як gradient clipping допомагає вирішити проблему вибухаючих градієнтів?

8. Що таке bidirectional RNN та в яких випадках вони корисні?

9. Чому класичні RNN погано справляються з довгостроковими залежностями?

10. Опишіть процес генерації послідовності за допомогою навченої RNN.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження рекурентної нейронної мережі

2. Мета

Зрозуміти принципи роботи RNN, реалізувати базову архітектуру, освоїти BPTT, дослідити проблеми навчання та методи їх вирішення.

3. Завдання

- Реалізувати базову RNN комірку

- Імплементувати BPTT алгоритм

- Додати gradient clipping

- Створити функції для різних типів задач

- Візуалізувати процес навчання та приховані стани

- Порівняти з LSTM/GRU

4. Опис

Студенти створюють повну реалізацію RNN “з нуля”, що дозволяє детально зрозуміти механізми роботи рекурентних мереж та проблеми їх навчання.

5. Порядок виконання

1. Реалізувати RNN cell з forward pass

2. Імплементувати розгортання в часі

3. Створити BPTT для обчислення градієнтів

4. Додати gradient clipping та різні методи ініціалізації

5. Реалізувати функції для генерації тексту

6. Створити візуалізацію градієнтів та активацій

7. Навчити на простому датасеті (char-level)

8. Експериментувати з гіперпараметрами

6. Результат

Функціональна реалізація RNN з можливістю навчання на текстових даних, візуалізацією внутрішніх процесів та аналізом проблем навчання.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас RNNCell з методами forward та backward

- Реалізація BPTT алгоритму

- Функції для різних типів задач

- Утиліти для обробки послідовностей

- Візуалізація навчання

2. Продемонструвати роботу додатку:

- Навчання character-level моделі

- Генерація тексту з різною температурою

- Візуалізація норм градієнтів

- Аналіз впливу gradient clipping

- Порівняння різних функцій активації

3. Бути готовим відповісти на питання:

- Як довжина послідовності впливає на навчання?
- Чому важлива правильна ініціалізація рекурентних ваг?
- Як діагностувати проблеми з градієнтами?
- Які обмеження має базова RNN архітектура?

Тести

1. Прихований стан RNN оновлюється на основі:
1) Тільки поточного входу
2) Поточного входу та попереднього прихованого стану
3) Всіх попередніх входів безпосередньо
4) Тільки попереднього прихованого стану
2. BPTT розшифровується як:
1) Backward Propagation Through Training
2) Backpropagation Through Time
3) Bidirectional Propagation Through Time
4) Batch Propagation Through Time
3. Основна проблема класичних RNN:
1) Велика кількість параметрів
2) Неможливість обробки тексту
3) Зникаючі/вибухаючі градієнти
4) Повільний inference
4. Gradient clipping обмежує:
1) Значення ваг
2) Норму градієнтів
3) Розмір прихованого стану
4) Довжину послідовності
5. Для задачі many-to-one використовується:
1) Вихід на кожному кроці
2) Тільки перший вихід
3) Тільки останній вихід
4) Усереднення всіх виходів

Відповіді:

1-2,

2-2,

3-3,

4-2,

5-3

Тести підвищеної складності

1. Чому ортогональна ініціалізація корисна для RNN?

1) Зменшує кількість параметрів

2) Зберігає норму градієнтів при проходженні через час

3) Прискорює збіжність

4) Спрощує обчислення

2. Truncated BPTT відрізняється від повного BPTT:

1) Використанням іншої функції втрат

2) Обмеженням кількості кроків для backpropagation

3) Відсутністю gradient clipping

4) Паралельною обробкою

3. Teacher forcing в RNN означає:

1) Використання більшої швидкості навчання

2) Подання справжніх виходів як входів на наступних кроках

3) Примусове обнулення градієнтів

4) Використання додаткових даних

4. Scheduled sampling вирішує проблему:

1) Повільного навчання

2) Exposure bias між навчанням та inference

3) Вибухаючих градієнтів

4) Перенавчання

5. Peephole connections в LSTM дозволяють:

1) Прискорити навчання

2) Гейтам мати прямий доступ до cell state

3) Зменшити кількість параметрів

4) Покращити паралелізацію

6. Variational dropout в RNN:

1) Застосовує різні маски dropout на кожному кроці

2) Використовує одну маску для всієї послідовності

3) Не застосовується до рекурентних зв’язків

4) Працює тільки для LSTM

7. Layer normalization краща за batch normalization для RNN тому що:

1) Швидше обчислюється

2) Не залежить від розміру батчу та довжини послідовності

3) Має менше параметрів

4) Простіша реалізація

8. Attention механізм в seq2seq дозволяє:

1) Прискорити навчання

2) Декодеру звертатися до всіх станів енкодера

3) Зменшити розмір моделі

4) Уникнути gradient clipping

9. CTC loss використовується для:

1) Класифікації послідовностей

2) Навчання без вирівнювання вхід-вихід

3) Регуляризації

4) Покращення градієнтів

10. Reservoir computing відрізняється від звичайних RNN:

1) Відсутністю прихованого стану

2) Фіксованими випадковими рекурентними вагами

3) Використанням CNN замість RNN

4) Обов’язковим використанням LSTM

Відповіді:

1. 2 - Ортогональні матриці зберігають норму при множенні

2. 2 - TBPTT обмежує глибину backpropagation для ефективності

3. 2 - При навчанні використовуються справжні, а не передбачені виходи

4. 2 - Поступовий перехід від teacher forcing до використання власних передбачень

5. 2 - Peephole дозволяє гейтам “заглядати” в cell state

6. 2 - Одна маска dropout для всієї послідовності для консистентності

7. 2 - LN нормалізує по features, не залежить від інших прикладів

8. 2 - Attention дозволяє селективний доступ до всієї вхідної послідовності

9. 2 - CTC дозволяє навчання на невирівняних послідовностях

10. 2 - Тільки вихідні ваги навчаються, рекурентні залишаються випадковими

Завдання для самостійної роботи

1. Реалізуйте двонаправлену RNN та порівняйте з однонаправленою версією на задачі POS tagging.

2. Створіть ансамбль RNN з різними архітектурами для покращення результатів.

3. Імплементуйте RNN з dropout для послідовностей.

4. Дослідіть вплив різних методів ініціалізації на швидкість збіжності та фінальну якість.

5. Реалізуйте механізм уваги (attention) для покращення якості на довгих послідовностях.

6. Створіть візуалізацію того, яка інформація зберігається в прихованому стані.

7. Імплементуйте ієрархічну RNN для обробки документів (речення → параграфи → документ).

8. Розробіть методи регуляризації специфічні для RNN (weight tying, activation regularization).

9. Реалізуйте continuous-time RNN для нерегулярно семплованих часових рядів.

10. Створіть інтерпретований варіант RNN з можливістю пояснення прийнятих рішень.

Теми лабораторних робіт

1. Базова реалізація RNN для character-level моделювання мови.

2. Порівняльний аналіз vanilla RNN, LSTM та GRU на різних задачах.

3. Музична генерація з використанням RNN: від MIDI до аудіо.

4. Sentiment analysis з візуалізацією внутрішніх представлень.

5. Seq2seq модель для простого машинного перекладу.

6. Time series прогнозування з uncertainty estimation.

7. Bidirectional RNN для named entity recognition.

8. RNN-based система для автодоповнення коду.

9. Multimodal RNN: об'єднання тексту та зображень.

10. Online навчання RNN для адаптивної обробки потокових даних.

Архітектура трансформера

Теоретичні відомості

Трансформер (Transformer) є революційною архітектурою нейронних мереж, представленою в статті "Attention is All You Need" у 2017 році. Ця архітектура повністю відмовляється від рекурентних та згорткових шарів, покладаючись виключно на механізм самоуваги (self-attention) для моделювання залежностей у послідовностях. Трансформер став основою для сучасних моделей обробки природної мови, таких як BERT, GPT, та багатьох інших, і знайшов застосування далеко за межами NLP.

Мотивація та основні принципи

Традиційні підходи до обробки послідовностей мали кілька фундаментальних обмежень:

1. Послідовна природа RNN - неможливість паралелізації обчислень, оскільки кожен крок залежить від попереднього, що призводить до повільного навчання на довгих послідовностях.

2. Обмежена довжина контексту - RNN та LSTM страждають від проблеми забування інформації при обробці довгих послідовностей.

3. Фіксоване вікно в CNN - згорткові мережі мають обмежене рецептивне поле та потребують багатьох шарів для моделювання довгих залежностей.

Трансформер вирішує ці проблеми через механізм самоуваги, який дозволяє кожній позиції в послідовності безпосередньо взаємодіяти з усіма іншими позиціями, забезпечуючи: - Повну паралелізацію обчислень -

Прямий доступ до будь-якої частини контексту - Ефективне моделювання довгих залежностей

Механізм самоуваги (Self-Attention)

Самоувага є серцем архітектури трансформера. Для кожного елемента послідовності механізм обчислює, наскільки важливі всі інші елементи для його представлення.

Математично, для вхідної послідовності $X \in \mathbb{R}^{n \times d}$, де n - довжина послідовності, d - розмірність представлення, самоувага обчислюється через три лінійні проекції:

Запити (Queries): $Q = XW^Q$, де $W^Q \in \mathbb{R}^{d \times d_k}$ Ключі (Keys): $K = XW^K$, де $W^K \in \mathbb{R}^{d \times d_k}$ Значення (Values): $V = XW^V$, де $W^V \in \mathbb{R}^{d \times d_v}$

Механізм уваги обчислюється як:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

де: - QK^T обчислює схожість між всіма парами позицій - Ділення на $\sqrt{d_k}$ стабілізує градієнти - Softmax нормалізує ваги уваги для кожної позиції - Множення на V агрегує інформацію згідно з вагами уваги

Багатоголова увага (Multi-Head Attention)

Замість використання однієї функції уваги, трансформер використовує багатоголову увагу, яка дозволяє моделі одночасно звертати увагу на інформацію з різних підпросторів представлення:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

де кожна голова обчислюється як:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Параметри: - h - кількість голів уваги - $W_i^Q \in \mathbb{R}^{d \times d_k}$, $W_i^K \in \mathbb{R}^{d \times d_k}$, $W_i^V \in \mathbb{R}^{d \times d_v}$ - $W^O \in \mathbb{R}^{hd_v \times d}$ - матриця вихідної проекції - Зазвичай $d_k = d_v = d/h$

Позиційне кодування

Оскільки механізм уваги сам по собі не враховує порядок елементів у послідовності, трансформер використовує позиційне кодування для додавання інформації про позицію:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

де: - pos - позиція в послідовності - i - індекс розмірності - d - розмірність моделі

Це кодування має властивості: - Унікальне представлення для кожної позиції - Можливість екстраполяції на довші послідовності - Відносні позиції можуть бути виражені через лінійні комбінації

Архітектура енкодера

Енкодер трансформера складається з N ідентичних шарів, кожен з яких містить два підшари:

1. Багатоголова самоувага

2. Позиційно-орієнтована повнозв'язна мережа (FFN)

Кожен підшар має залишкове з'єднання та нормалізацію шару:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

FFN складається з двох лінійних перетворень з ReLU активацією:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

де $W_1 \in \mathbb{R}^{d \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d}$, зазвичай $d_{ff} = 4d$.

Архітектура декодера

Декодер також складається з N шарів, але кожен шар має три підшари:

1. Маскована багатоголова самоувага - запобігає "підглядання" в майбутні позиції

2. Багатоголова увага енкодер-декодер - звертається до виходу енкодера

3. FFN - така ж, як в енкодері

Маскування в самоувазі досягається встановленням $-\infty$ для недозволених зв'язків перед softmax.

Навчання та inference

Навчання відбувається паралельно для всієї послідовності з використанням teacher forcing та маскування для запобігання витоку інформації з майбутнього.

Inference (генерація) відбувається авторегресивно: 1. Починаємо з початкового токена 2. Генеруємо наступний токен 3. Додаємо його до контексту 4. Повторюємо до досягнення кінцевого токена або максимальної довжини

Псевдокод трансформера

```
функція ScaledDotProductAttention(Q, K, V, mask=null)
    // Q: [batch, heads, seq_len, d_k]
    // K: [batch, heads, seq_len, d_k]
    // V: [batch, heads, seq_len, d_v]

    d_k = розмір(K, -1)

    // Обчислення scores
    scores = матричне_множення(Q, транспонувати(K, -2, -1)) / sqrt(d_
k)

    // Застосування маски (якщо є)
    якщо mask != null тоді
        scores = scores + mask * (-1e9)

    // Softmax по останньому виміру
    attention_weights = softmax(scores, dim=-1)

    // Застосування dropout (під час навчання)
    attention_weights = dropout(attention_weights, p=0.1)

    // Зважене усереднення значень
    output = матричне_множення(attention_weights, V)

    повернути output, attention_weights

функція MultiHeadAttention(x, num_heads, d_model, mask=null)
    // x: [batch, seq_len, d_model]

    batch_size = розмір(x, 0)
    seq_len = розмір(x, 1)
    d_k = d_model / num_heads
    d_v = d_model / num_heads

    // Лінійні проекції для Q, K, V
    Q = лінійний_шар(x, d_model, d_model)  // W_Q
    K = лінійний_шар(x, d_model, d_model)  // W_K
    V = лінійний_шар(x, d_model, d_model)  // W_V

    // Розділення на голови
    Q = перефомувати(Q, [batch_size, seq_len, num_heads, d_k])
    Q = транспонувати(Q, 1, 2)  // [batch, heads, seq_len, d_k]

    K = перефомувати(K, [batch_size, seq_len, num_heads, d_k])
    K = транспонувати(K, 1, 2)
```

```
    V = перефомувати(V, [batch_size, seq_len, num_heads, d_v])
    V = транспонувати(V, 1, 2)

    // Застосування уваги
    attention_output, attention_weights = ScaledDotProductAttention(Q
, K, V, mask)

    // Об'єднання голів
    attention_output = транспонувати(attention_output, 1, 2)
    attention_output = перефомувати(attention_output, [batch_size, se
q_len, d_model])

    // Фінальна лінійна проекція
    output = лінійний_шар(attention_output, d_model, d_model)  // W_O

    повернути output, attention_weights

функція PositionalEncoding(seq_len, d_model)
    // Створення позиційного кодування

    PE = нулі([seq_len, d_model])
    position = аранж(0, seq_len).розширити_виміри(1)

    div_term = exp(аранж(0, d_model, 2) * -(log(10000.0) / d_model))

    PE[:, 0::2] = sin(position * div_term)  // парні індекси
    PE[:, 1::2] = cos(position * div_term)  // непарні індекси

    повернути PE

функція TransformerEncoderLayer(x, d_model, num_heads, d_ff, dropout_
rate)
    // x: [batch, seq_len, d_model]

    // Підшар 1: Multi-Head Self-Attention
    attn_output, _ = MultiHeadAttention(x, num_heads, d_model)
    attn_output = dropout(attn_output, dropout_rate)
    x = layer_norm(x + attn_output)  // Залишкове з'єднання + LayerNo
rm

    // Підшар 2: Feed Forward
    ff_output = FFN(x, d_model, d_ff)
    ff_output = dropout(ff_output, dropout_rate)
    x = layer_norm(x + ff_output)  // Залишкове з'єднання + LayerNorm
```

```
    повернути x

функція FFN(x, d_model, d_ff)
    // Позиційно-орієнтована FFN
    x = лінійний_шар(x, d_model, d_ff)
    x = relu(x)
    x = dropout(x, 0.1)
    x = лінійний_шар(x, d_ff, d_model)
    повернути x

функція TransformerEncoder(x, num_layers, d_model, num_heads, d_ff, d
ropout_rate)
    // Додавання позиційного кодування
    seq_len = розмір(x, 1)
    pos_encoding = PositionalEncoding(seq_len, d_model)
    x = x + pos_encoding

    // Проходження через шари енкодера
    для i = 0 до num_layers-1 виконати
        x = TransformerEncoderLayer(x, d_model, num_heads, d_ff, drop
out_rate)

    повернути x

функція TransformerDecoderLayer(x, encoder_output, d_model, num_heads
, d_ff,
                                dropout_rate, look_ahead_mask, padding_
mask)
    // Підшар 1: Masked Multi-Head Self-Attention
    attn1_output, _ = MultiHeadAttention(x, num_heads, d_model, look_
ahead_mask)
    attn1_output = dropout(attn1_output, dropout_rate)
    x = layer_norm(x + attn1_output)

    // Підшар 2: Multi-Head Cross-Attention
    attn2_output, _ = CrossAttention(x, encoder_output, encoder_outpu
t,
                                     num_heads, d_model, padding_mask)
    attn2_output = dropout(attn2_output, dropout_rate)
    x = layer_norm(x + attn2_output)

    // Підшар 3: Feed Forward
    ff_output = FFN(x, d_model, d_ff)
    ff_output = dropout(ff_output, dropout_rate)
```

```
    x = layer_norm(x + ff_output)

    повернути x

функція CreateLookAheadMask(size)
    // Створення маски для запобігання підглядання в майбутнє
    mask = трикутна_матриця(size, size)
    повернути mask  // 0 для дозволених позицій, 1 для заборонених

функція Transformer(src, tgt, num_layers, d_model, num_heads, d_ff,
                    src_vocab_size, tgt_vocab_size, dropout_rate)
    // src: вхідна послідовність
    // tgt: цільова послідовність (при навчанні)

    // Embedding шари
    src_embedding = embedding_шар(src, src_vocab_size, d_model)
    tgt_embedding = embedding_шар(tgt, tgt_vocab_size, d_model)

    // Масштабування embeddings
    src_embedding *= sqrt(d_model)
    tgt_embedding *= sqrt(d_model)

    // Енкодер
    encoder_output = TransformerEncoder(src_embedding, num_layers,
                                        d_model, num_heads, d_ff, dropo
ut_rate)

    // Декодер
    look_ahead_mask = CreateLookAheadMask(розмір(tgt, 1))

    decoder_output = tgt_embedding
    для i = 0 до num_layers-1 виконати
        decoder_output = TransformerDecoderLayer(
            decoder_output, encoder_output, d_model, num_heads,
            d_ff, dropout_rate, look_ahead_mask, null)

    // Фінальний лінійний шар для прогнозування
    output = лінійний_шар(decoder_output, d_model, tgt_vocab_size)

    повернути output

// Функція для генерації (inference)
функція GenerateSequence(model, src, max_length, start_token, end_tok
en)
    // Кодування вхідної послідовності
```

```
    encoder_output = model.encode(src)

    // Початкова послідовність декодера
    output_sequence = [start_token]

    для i = 0 до max_length-1 виконати
        // Створення вхідного тензора для декодера
        tgt = тензор(output_sequence)

        // Отримання прогнозів
        predictions = model.decode(tgt, encoder_output)

        // Вибір наступного токена (greedy або sampling)
        next_token = argmax(predictions[-1])  // greedy decoding

        output_sequence.додати(next_token)

        якщо next_token == end_token тоді
            перервати

    повернути output_sequence
```

Оптимізації та варіації

1. Flash Attention - оптимізована реалізація, що зменшує використання пам'яті з $O(n^2)$ до $O(n)$.

2. Sparse Attention - обчислення уваги лише для підмножини позицій для довгих послідовностей.

3. Relative Position Encoding - кодування відносних, а не абсолютних позицій.

4. Pre-LayerNorm vs Post-LayerNorm - розміщення layer normalization перед або після підшарів.

Приклад застосування

Розглянемо спрощений приклад роботи трансформера для задачі перекладу короткої фрази.

Задача: Переклад "I love AI" → "Я люблю ШІ"

Словники:

- Англійський: {PAD: 0, START: 1, END: 2, I: 3, love: 4, AI: 5}

- Український: {PAD: 0, START: 1, END: 2, Я: 3, люблю: 4, ШІ: 5}

Параметри моделі (спрощені):

- d_model = 4

- num_heads = 2

- num_layers = 1

Крок 1: Кодування вхідної послідовності

Вхід: [3, 4, 5] (I love AI)

Embeddings (випадкові для прикладу):

```
I:    [0.2, -0.1, 0.3, 0.1]
love: [0.1, 0.4, -0.2, 0.3]
AI:   [-0.1, 0.2, 0.1, 0.4]
```

Позиційне кодування:

```
pos_0: [0.0, 1.0, 0.0, 1.0]
pos_1: [0.84, 0.54, 0.01, 1.0]
pos_2: [0.91, -0.42, 0.02, 1.0]
```

Вхід енкодера = Embeddings + Позиційне кодування

Крок 2: Self-Attention в енкодері

Для голови 1 (d_k = 2):

```
Q = [[0.1, 0.2],    K = [[0.2, 0.1],    V = [[0.3, 0.1],
     [0.3, 0.1],         [0.1, 0.3],         [0.2, 0.2],
     [0.2, 0.2]]         [0.2, 0.2]]         [0.1, 0.3]]
```

Attention scores = QK^T / √2:

```
     I    love  AI
I   [0.03, 0.04, 0.04]
love[0.05, 0.08, 0.06]
AI  [0.04, 0.06, 0.06]
```

Після softmax:

```
     I     love   AI
I   [0.33, 0.34, 0.33]
love[0.32, 0.36, 0.32]
AI  [0.32, 0.34, 0.34]
```

Attention output = weights × V

Крок 3: Feed-Forward Network

Застосування двошарової мережі з розширенням до d_ff = 16 та звуженням назад до d_model = 4.

Крок 4: Декодування

Початковий вхід декодера: [START]

Ітерація 1:

- Вхід: [1] (START)
- Cross-attention з encoder output
- Прогноз: “Я” (token 3)

Ітерація 2:

- Вхід: [1, 3] (START, Я)
- Masked self-attention (не може бачити майбутнє)
- Cross-attention
- Прогноз: “люблю” (token 4)

Ітерація 3:

- Вхід: [1, 3, 4] (START, Я, люблю)
- Прогноз: “ШІ” (token 5)

Ітерація 4:

- Вхід: [1, 3, 4, 5]
- Прогноз: END (token 2)
- Завершення генерації

Візуалізація уваги:

Для фрази “I love AI”, матриця уваги може виглядати так:

```
        I    love   AI
I     0.7   0.2   0.1
love  0.3   0.5   0.2
AI    0.2   0.3   0.5
```

Це показує, що “I” найбільше звертає увагу на себе, “love” розподіляє увагу між всіма словами, а “AI” фокусується на собі та дієслові.

Область застосування

Трансформери революціонізували багато областей машинного навчання:

Обробка природної мови:

- Мовні моделі (GPT, BERT, T5)
- Машинний переклад
- Відповіді на питання
- Узагальнення тексту
- Генерація тексту

Комп’ютерний зір.

Мультимодальні задачі:

- CLIP (зображення + текст)
- DALL-E (текст → зображення)
- Flamingo (зображення + текст → текст)

Обробка аудіо:

- Wav2Vec2 - Whisper - Music generation

Переваги та недоліки

Переваги

Паралелізація навчання:

- На відміну від RNN, всі позиції обробляються одночасно
- Значне прискорення навчання
- Ефективне використання GPU/TPU

Довгі залежності:

- Пряме з’єднання між будь-якими позиціями
- Відсутність проблеми зникаючих градієнтів
- Ефективне моделювання контексту

Гнучкість архітектури:

- Легко адаптується для різних задач
- Модульна структура
- Можливість pre-training та transfer learning

Інтерпретованість:

- Візуалізація attention weights
- Розуміння того, на що модель звертає увагу
- Аналіз внутрішніх представлень

Масштабованість:

- Продуктивність зростає з розміром моделі
- Можливість розподіленого навчання
- Ефективні техніки квантизації

Недоліки

Квадратична складність:

- $O(n^2)$ по довжині послідовності
- Обмеження на максимальну довжину
- Високе споживання пам’яті

Потреба в великих даних:

- Мільйони або мільярди параметрів
- Схильність до перенавчання на малих датасетах
- Дорогий pre-training

Обчислювальні ресурси:

- Високі вимоги до GPU/TPU
- Значне енергоспоживання
- Дорога інфраструктура

Відсутність індуктивних упереджень:

- Не має вбудованого розуміння локальності (як CNN)
- Не має вбудованої рекурентності (як RNN)
- Потребує більше даних для навчання базових патернів

Складність оптимізації:

- Чутливість до learning rate
- Потреба в складних розкладах навчання
- Нестабільність при навчанні великих моделей

Контрольні запитання

1. Поясніть основну ідею механізму самоуваги та чому він ефективний для моделювання послідовностей.
2. Яка роль позиційного кодування в архітектурі трансформера?
3. Чим відрізняється багатоголова увага від одноголової та які переваги вона надає?
4. Опишіть різницю між енкодером та декодером в архітектурі трансформера.
5. Як працює маскування в декодері та чому воно необхідне?
6. Поясніть роль залишкових з’єднань та layer normalization в трансформері.

7. Чому трансформер має квадратичну складність по довжині послідовності?

8. Які основні відмінності між трансформером та RNN для обробки послідовностей?

9. Як відбувається процес генерації тексту в трансформері?

10. Опишіть основні виклики при навчанні великих трансформерів.

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження архітектури трансформера

2. Мета

Глибоко зрозуміти принципи роботи трансформера, реалізувати ключові компоненти, навчитися застосовувати для різних задач.

3. Завдання

- Реалізувати механізм scaled dot-product attention

- Створити багатоголову увагу

- Імплементувати позиційне кодування

- Побудувати повну архітектуру трансформера

- Візуалізувати attention patterns

- Порівняти з RNN/LSTM

4. Опис

Студенти створюють власну реалізацію трансформера, починаючи з базових компонентів, що дозволяє зрозуміти всі аспекти цієї архітектури.

5. Порядок виконання

1. Реалізувати функцію обчислення уваги з масштабуванням

2. Розширити до багатоголової уваги

3. Створити позиційне кодування

4. Реалізувати encoder layer з FFN

5. Додати decoder layer з маскуванням

6. Зібрати повний трансформер

7. Реалізувати навчання та inference

8. Візуалізувати та аналізувати результати

6. Результат

Робоча реалізація трансформера з можливістю навчання на задачах послідовність-до-послідовності, візуалізацією механізмів уваги.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас MultiHeadAttention
- Клас TransformerEncoderLayer
- Клас TransformerDecoderLayer
- Повна модель Transformer
- Функції для навчання та генерації

2. Продемонструвати роботу додатку:

- Навчання на простій задачі перекладу
- Візуалізація attention weights
- Генерація послідовностей
- Порівняння швидкості з RNN
- Аналіз впливу кількості голів

3. Бути готовим відповісти на питання:

- Як змінюється якість з кількістю шарів та голів?
- Який вплив має розмір моделі на продуктивність?
- Як інтерпретувати візуалізації уваги?
- Які обмеження має реалізована модель?

Тести

1. Основна інновація трансформера:

1) Використання LSTM замість RNN

2) Відмова від рекурентності на користь уваги

3) Додавання згорткових шарів

4) Використання більших моделей

2. Складність механізму уваги по довжині послідовності:

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) $O(n^3)$

3. Позиційне кодування необхідне тому що:

1) Покращує якість моделі

2) Увага сама по собі не враховує порядок

3) Зменшує кількість параметрів

4) Прискорює навчання

4. В багатоголовій увазі кожна голова:

1) Має власні параметри проекції

2) Використовує ті ж параметри

3) Обробляє різні частини послідовності

4) Має різні функції активації

5. Маскування в декодері запобігає:

1) Перенавчанню

2) Підглядання в майбутні позиції

3) Вибуху градієнтів

4) Повільному навчанню

Відповіді:

1-2,

2-3,

3-2,

4-1,

5-2

Тести підвищеної складності

1. Flash Attention оптимізує трансформер через:

1) Використання sparse patterns

2) Tiling та переупорядкування обчислень для кращого використання пам’яті

3) Квантизацію ваг

4) Зменшення кількості голів

2. У Vision Transformer зображення розбивається на патчі тому що:

1) Це покращує якість

2) Трансформер працює з послідовностями токенів

3) Зменшує кількість параметрів

4) Прискорює навчання

3. Rotary Position Embedding (RoPE) відрізняється від класичного тим що:

1) Не потребує навчання

2) Кодує відносні позиції через обертання в комплексному просторі

3) Працює тільки для коротких послідовностей

4) Використовує згорткові шари

4. ALiBi (Attention with Linear Biases) вирішує проблему:

1) Квадратичної складності

2) Екстраполяції на довші послідовності

3) Нестабільності навчання

4) Великої кількості параметрів

5. Mixture of Experts (MoE) в трансформерах:

1) Використовує кілька моделей одночасно

2) Динамічно активує підмножину параметрів для кожного токена

3) Завжди покращує якість

4) Зменшує кількість параметрів

6. Gradient checkpointing в трансформерах:

1) Зберігає всі активації

2) Торгує обчисленнями за пам’ять через переобчислення

3) Прискорює навчання

4) Покращує якість моделі

7. Чому Pre-LN (Pre-Layer Normalization) стабільніша за Post-LN?

1) Має менше параметрів

2) Градієнти не накопичують масштаб через багато шарів

3) Працює швидше

4) Краща якість моделі

8. Sparse Transformer з фіксованими патернами уваги:

1) Завжди гірший за повну увагу

2) Може апроксимувати повну увагу з меншою складністю

3) Не працює для NLP

4) Потребує спеціального обладнання

9. KV-кешування при генерації:

1) Зберігає всі активації

2) Зберігає ключі та значення попередніх кроків

3) Уповільнює генерацію

4) Збільшує якість

10. Чому BERT використовує лише енкодер, а GPT - лише декодер?

1) Це випадковий вибір

2) Різні задачі: двонаправлене розуміння vs авторегресивна генерація

3) Декодер повільніший

4) Енкодер має більше параметрів

Відповіді:

1. 2 - Flash Attention оптимізує доступ до пам’яті через tiling

2. 2 - Патчі стають токенами послідовності для трансформера

3. 2 - RoPE кодує позиції через обертання в просторі представлень

4. 2 - ALiBi додає лінійні біаси для кращої екстраполяції

5. 2 - MoE активує експертів залежно від вхідних даних

6. 2 - Переобчислює активації під час backward pass

7. 2 - Нормалізація перед шаром стабілізує градієнти

8. 2 - Фіксовані патерни можуть ефективно апроксимувати повну увагу

9. 2 - Кешування KV запобігає переобчисленню для згенерованих токенів

10. 2 - Архітектури оптимізовані для різних типів задач

Завдання для самостійної роботи

1. Реалізуйте різні види позиційного кодування (learned, rotary, ALiBi) та порівняйте їх ефективність.

2. Створіть sparse attention patterns та дослідіть вплив на якість та швидкість.

3. Імплементуйте Flash Attention для ефективного використання пам’яті.

4. Розробіть методи візуалізації та інтерпретації різних шарів трансформера.

5. Реалізуйте knowledge distillation для стиснення великого трансформера.

6. Створіть гібридну архітектуру, що комбінує трансформер з CNN або RNN.

7. Дослідіть різні стратегії tokenization та їх вплив на продуктивність.

8. Імплементуйте ефективні методи fine-tuning (LoRA, adapters).

9. Розробіть методи для роботи з довгими послідовностями (Longformer, BigBird).

10. Створіть benchmark для порівняння різних оптимізацій трансформера.

Теми лабораторних робіт

1. Базова реалізація трансформера для задачі машинного перекладу.

2. Vision Transformer: адаптація для класифікації зображень.

3. BERT-style pre-training: masked language modeling.

4. GPT-style генерація тексту з різними стратегіями декодування.

5. Multimodal transformer: об’єднання тексту та зображень.

6. Efficient transformers: реалізація та порівняння Linformer, Performer.

7. Fine-tuning стратегії: повне, LoRA, prefix tuning.

8. Attention visualization toolkit для аналізу моделей.

9. Streaming transformer для real-time обробки.

10. Transformer compression: pruning, quantization, distillation.

Великі мовні моделі (LLM)

Теоретичні відомості

Великі мовні моделі (Large Language Models, LLM) представляють собою клас нейронних мереж на основі архітектури трансформера, навчених на величезних обсягах текстових даних для розуміння та генерації природної мови. Ці моделі, що містять від мільйонів до трильйонів параметрів, демонструють емерджентні здібності - властивості, які виникають при

досягненні певного масштабу та не спостерігаються в менших моделях. LLM фундаментально змінили підходи до обробки природної мови, демонструючи здатність до виконання широкого спектру завдань без спеціалізованого навчання.

Великі мовні моделі базуються на кількох фундаментальних принципах:

1. Авторегресивне моделювання - LLM навчаються передбачати наступний токен у послідовності, враховуючи всі попередні токени. Це формулюється як задача максимізації ймовірності:

$$P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P\left(x_i | x_1, x_2, \ldots, x_{i-1}\right)$$

де x_i - токен на позиції i, а умовна ймовірність $P(x_i|x_1, \ldots, x_{i-1})$ моделюється нейронною мережею.

2. Самонавчання (Self-supervised learning) - моделі навчаються на неанотованих текстових даних, використовуючи сам текст як джерело навчальних сигналів. Це дозволяє використовувати величезні обсяги доступного тексту з інтернету, книг, статей тощо.

3. Контекстне навчання (In-context learning) - здатність виконувати нові завдання, отримуючи лише кілька прикладів у текстовому промпті, без оновлення параметрів моделі.

Архітектура LLM

Сучасні LLM базуються на декодерній частині трансформера з модифікаціями для покращення ефективності та якості:

Базова архітектура складається з:

- Вбудованого шару для перетворення токенів у векторні представлення
- Позиційного кодування (абсолютного або відносного)
- Послідовності декодерних блоків трансформера
- Вихідного шару для прогнозування наступного токена

Кожен декодерний блок містить:

- Маскований механізм самоуваги
- Нормалізацію шару (pre-norm або post-norm)
- Позиційно-орієнтовану мережу прямого поширення (FFN)

- Залишкові з’єднання

Модифікації архітектури:

1. RMSNorm замість LayerNorm для ефективності:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}} \cdot \gamma$$

2. SwiGLU активація замість ReLU в FFN:

$$\text{SwiGLU}(x, W, V) = \text{Swish}(xW) \otimes xV$$

де $\text{Swish}(x) = x \cdot \sigma(x)$

3. Rotary Position Embeddings (RoPE) для кращого моделювання позицій:

$$f_q(x_m, m) = x_m e^{im\theta}$$

4. Multi-Query або Grouped-Query Attention для ефективності inference.

Процес навчання LLM

Навчання великих мовних моделей відбувається в кілька етапів:

Pre-training - основне навчання на великому корпусі текстів:

1. Підготовка даних:

- Збір текстів з різних джерел (веб, книги, код, наукові статті)
- Очищення та дедуплікація
- Токенізація з використанням subword алгоритмів (BPE, SentencePiece)

2. Оптимізація:

- Функція втрат: крос-ентропія для next token prediction

$$L = -\sum_{i=1}^{N} \log P(x_i | x_{<i})$$

- Оптимізатор: AdamW з косинусним розкладом learning rate
- Gradient accumulation для ефективних великих батчів
- Mixed precision training (FP16/BF16)

3. Масштабування:

- Data parallelism: розподіл батчів між GPU
- Model parallelism: розподіл шарів моделі
- Pipeline parallelism: конвеєрна обробка

- Tensor parallelism: розподіл тензорних операцій

Instruction Tuning - донавчання для слідування інструкціям:

- Навчання на парах (інструкція, відповідь)

- Покращення здатності виконувати конкретні завдання

- Використання високоякісних анотованих даних

Вирівнювання через навчання з підкріпленням на основі зворотного зв'язку від людини (RLHF) (Reinforcement Learning from Human Feedback):

1. Кероване тонке налаштування (supervised fine-tuning) (SFT) на демонстраціях

2. Навчання моделі винагород на людських преференціях

3. Оптимізація політики через PPO або DPO

Токенізація та словники

LLM використовують токенізацію на підсловах для балансу між розміром словника та гранулярністю:

Кодування методом попарного об'єднання байтів (BPE):

- Ітеративне об'єднання найчастіших пар символів

- Створення словника фіксованого розміру

- Можливість кодування будь-якого тексту

Токенізація SentencePiece (SentencePiece):

- Навчання безпосередньо на сирому тексті

- Підтримка різних алгоритмів (BPE, Unigram)

- Детерміністична токенізація

Розмір словника впливає на:

- Ефективність представлення (compression rate)

- Розмір вбудованої матриці

- Швидкість висновку

Механізми уваги в LLM

Сучасні LLM використовують оптимізовані варіанти уваги:

1. Flash Attention:

- Tiling для ефективного використання GPU пам'яті

- IO-aware алгоритм

- Лінійна складність по пам’яті

2. Sliding Window Attention:

- Обмеження уваги локальним вікном

- Періодичні глобальні токени

- Ефективність для довгих контекстів

3. Sparse Attention:

- Фіксовані або навчені патерни розрідженості

- Зменшення квадратичної складності

- Збереження якості моделювання

Емерджентні властивості

При досягненні певного масштабу LLM демонструють здібності, не присутні в менших моделях:

1. Few-shot learning - здатність вирішувати нові завдання з кількома прикладами:

- Без градієнтних оновлень

- Через pattern matching в контексті

- Покращується з розміром моделі

2. Chain-of-thought reasoning - покрокове міркування:

- Декомпозиція складних задач

- Проміжні кроки покращують точність

- Може бути викликане промптингом

3. Emergent abilities включають:

- Арифметичні операції

- Логічне виведення

- Розуміння каузальності

- Творче письмо

- Програмування

Псевдокод LLM

```
функція LLM_Model(config)
    // config містить: n_layers, d_model, n_heads, vocab_size, max_le
```

```
ngth

    // Ініціалізація компонентів
    token_embeddings = EmbeddingLayer(config.vocab_size, config.d_mod
el)
    position_embeddings = RotaryEmbeddings(config.d_model, config.max
_length)

    layers = []
    для i = 0 до config.n_layers-1 виконати
        layer = TransformerDecoderLayer(
            d_model=config.d_model,
            n_heads=config.n_heads,
            d_ff=4*config.d_model,
            dropout=0.1
        )
        layers.додати(layer)

    final_norm = RMSNorm(config.d_model)
    output_projection = Linear(config.d_model, config.vocab_size)

    повернути {
        embeddings: token_embeddings,
        pos_encoding: position_embeddings,
        layers: layers,
        final_norm: final_norm,
        output: output_projection
    }

функція LLM_Forward(model, input_ids, attention_mask=null)
    // input_ids: [batch_size, seq_len]

    batch_size, seq_len = розміри(input_ids)

    // Token embeddings
    x = model.embeddings(input_ids)  // [batch, seq_len, d_model]

    // Позиційне кодування
    positions = аранж(seq_len)
    x = model.pos_encoding(x, positions)

    // Створення каузальної маски
    causal_mask = створити_каузальну_маску(seq_len)
    якщо attention_mask != null тоді
        causal_mask = causal_mask * attention_mask
```

```
    // Проходження через шари
    для layer в model.layers виконати
        x = layer(x, causal_mask)

    // Фінальна нормалізація
    x = model.final_norm(x)

    // Проекція на словник
    logits = model.output(x)  // [batch, seq_len, vocab_size]

    повернути logits

функція TransformerDecoderLayer(x, mask)
    // Masked Multi-Head Self-Attention
    residual = x
    x = RMSNorm(x)
    x = MultiQueryAttention(x, x, x, mask)
    x = dropout(x, 0.1)
    x = x + residual

    // Feed-Forward Network
    residual = x
    x = RMSNorm(x)
    x = SwiGLU_FFN(x)
    x = dropout(x, 0.1)
    x = x + residual

    повернути x

функція SwiGLU_FFN(x, d_model, d_ff)
    // SwiGLU activation для FFN
    gate = Linear(x, d_model, d_ff)
    up = Linear(x, d_model, d_ff)

    gate = swish(gate)  // swish(x) = x * sigmoid(x)
    x = gate * up

    x = Linear(x, d_ff, d_model)
    повернути x

функція Train_LLM(model, train_data, config)
    optimizer = AdamW(model.parameters(), lr=config.lr, weight_decay=
0.1)
    scheduler = CosineSchedule(optimizer, warmup_steps=config.warmup)
```

```
    для epoch = 1 до config.epochs виконати
        для batch в DataLoader(train_data, batch_size=config.batch_si
ze) виконати
            input_ids = batch.input_ids
            labels = batch.labels  // зсунуті input_ids

            // Forward pass
            logits = LLM_Forward(model, input_ids)

            // Обчислення втрат
            loss = CrossEntropyLoss(
                перегляд(logits, -1, vocab_size),
                перегляд(labels, -1)
            )

            // Backward pass
            loss.backward()

            // Gradient clipping
            clip_grad_norm(model.parameters(), max_norm=1.0)

            // Оновлення параметрів
            optimizer.step()
            scheduler.step()
            optimizer.zero_grad()

            якщо крок % 100 == 0 тоді
                вивести "Loss:", loss.item()

функція Generate(model, prompt, max_length, temperature=1.0)
    // Токенізація промпту
    input_ids = tokenize(prompt)

    для i = довжина(input_ids) до max_length-1 виконати
        // Forward pass
        logits = LLM_Forward(model, input_ids)

        // Отримання логітів для останнього токена
        next_token_logits = logits[-1, -1, :]

        // Застосування температури
        next_token_logits = next_token_logits / temperature

        // Sampling стратегія
```

```
        якщо config.top_k > 0 тоді
            next_token_logits = top_k_filtering(next_token_logits, co
nfig.top_k)

        якщо config.top_p > 0 тоді
            next_token_logits = top_p_filtering(next_token_logits, co
nfig.top_p)

        // Семплювання наступного токена
        probs = softmax(next_token_logits)
        next_token = multinomial_sample(probs)

        // Додавання до послідовності
        input_ids = конкатенація(input_ids, next_token)

        // Перевірка на EOS токен
        якщо next_token == EOS_TOKEN тоді
            перервати

    повернути detokenize(input_ids)

// Функції для RLHF
функція RLHF_Training(model, reward_model, dataset)
    // Крок 1: SFT (Supervised Fine-tuning)
    sft_model = SupervisedFineTune(model, dataset.demonstrations)

    // Крок 2: Навчання моделі винагород
    reward_model = TrainRewardModel(dataset.comparisons)

    // Крок 3: PPO оптимізація
    policy = sft_model
    value_model = копія(sft_model)

    для episode в dataset.prompts виконати
        // Генерація відповіді
        response = Generate(policy, episode.prompt)

        // Отримання винагороди
        reward = reward_model(episode.prompt, response)

        // PPO update
        old_log_probs = обчислити_log_probs(policy, episode.prompt, r
esponse)
        new_log_probs = обчислити_log_probs(policy, episode.prompt, r
esponse)
```

```
        ratio = exp(new_log_probs - old_log_probs)
        clipped_ratio = clip(ratio, 1-ε, 1+ε)

        policy_loss = -min(ratio * advantages, clipped_ratio * advant
ages)
        value_loss = MSE(value_model(state), returns)

        total_loss = policy_loss + 0.5 * value_loss

        // Оновлення
        оновити_параметри(policy, total_loss)

    повернути policy
```

Оптимізації для ефективності

1. KV-Cache для прискорення генерації:

- Зберігання ключів та значень попередніх токенів
- Уникнення переобчислень при авторегресивній генерації
- Лінійне зростання пам’яті з довжиною

2. Quantization для зменшення розміру:

- INT8/INT4 квантизація ваг
- Quantization-aware training
- Dynamic quantization під час висновку

3. Model Sharding:

- Tensor parallelism: розподіл матричних операцій
- Pipeline parallelism: розподіл шарів
- Data parallelism: розподіл батчів

Промптинг та взаємодія

Ефективне використання LLM залежить від правильного промптингу:

Zero-shot prompting:

```
Завдання: Класифікуй настрій тексту.
Текст: "Цей фільм був просто чудовий!"
Відповідь:
```

Few-shot prompting:

```
Приклад 1: "Жахливий сервіс" → Негативний
Приклад 2: "Чудова їжа" → Позитивний
Текст: "Непоганий ресторан" →
```

Chain-of-thought:

```
Задача: Скільки буде 15% від 80?
Давай подумаємо крок за кроком:
1) 15% = 15/100 = 0.15
2) 0.15 × 80 = 12
Відповідь: 12
```

Приклад застосування

Розглянемо детальний приклад роботи спрощеної LLM для генерації тексту.

Модель:

Міні-LLM з 2 шарами, 4 головами уваги, d_model=128

Словник:

1000 токенів (включаючи спеціальні)

Завдання:

Продовження тексту “Машинне навчання це”

Крок 1: Токенізація

```
Вхід: "Машинне навчання це"
Токени: [256, 834, 15]
(256="Машинне", 834="навчання", 15="це")
```

Крок 2: Embeddings та позиційне кодування

```
Token embeddings:
[256] → [0.23, -0.14, 0.67, ..., 0.11]  # 128-вимірний вектор
[834] → [-0.31, 0.52, -0.08, ..., 0.45]
[15]  → [0.19, -0.27, 0.84, ..., -0.22]

Позиційне кодування (RoPE):
pos_0: застосування ротації з кутом 0
pos_1: застосування ротації з кутом θ
pos_2: застосування ротації з кутом 2θ
```

Крок 3: Проходження через шари

Шар 1, Голова 1 уваги:

```
Q = [[-0.15, 0.82], [0.34, -0.21], [0.67, 0.13]]
K = [[0.23, -0.45], [-0.12, 0.78], [0.55, -0.31]]
V = [[0.91, -0.17], [0.44, 0.62], [-0.28, 0.85]]

Attention scores (після масштабування та маскування):
```

```
     т0    т1    т2
т0 [[0.9,  0,    0  ],
т1  [0.4,  0.6,  0  ],
т2  [0.2,  0.3,  0.5]]
```

```
Output після зваженого усереднення V
```

FFN в шарі 1:

```
Вхід: [128-d вектори для кожного токена]
Розширення до 512-d через SwiGLU
Стиснення назад до 128-d
```

Аналогічно для шару 2

Крок 4: Фінальна проекція

```
Останній токен після всіх шарів: [0.43, -0.28, ..., 0.71]
Проекція на словник (1000 токенів):
logits = [2.3, -1.5, 0.8, ..., 3.7, -0.2, 4.1]
```

Крок 5: Семплювання наступного токена

```
Top-k (k=50) filtering: залишаємо 50 найімовірніших
Temperature (T=0.8): logits/0.8
Softmax: перетворення в ймовірності
Семплювання: токен 492 ("галузь")
```

Крок 6: Продовження генерації

```
Новий вхід: [256, 834, 15, 492]
Повторення кроків 2-5
Наступний токен: 673 ("штучного")
І так далі...
```

Фінальний результат: “Машинне навчання це галузь штучного інтелекту, яка вивчає алгоритми…”

Аналіз уваги

Візуалізація патернів уваги показує:

- “навчання” звертає увагу на “Машинне” (0.6) - “це” розподіляє увагу між попередніми словами

- Пізніші токени більше фокусуються на релевантному контексті

Область застосування

LLM знайшли широке застосування в багатьох сферах:

Природна мова:

- Чат-боти та віртуальні асистенти

- Автоматичне написання контенту

- Переклад та локалізація
- Узагальнення документів
- Відповіді на питання

Програмування:

- Генерація коду
- Відладка та рефакторинг
- Документування коду
- Code review
- Навчання програмуванню

Освіта:

- Персоналізовані навчальні системи
- Генерація навчальних матеріалів
- Оцінювання та feedback
- Мовне навчання

Наукові дослідження:

- Літературний огляд
- Генерація гіпотез
- Аналіз даних
- Написання статей

Приклади застосування

Приклад 1: GitHub Copilot для програмування

Приклад 2: ChatGPT в освіті

Приклад 3: Claude для бізнес-аналітики

Переваги та недоліки

Переваги

Універсальність:

- Одна модель для багатьох задач
- Адаптація через промптинг
- Transfer learning можливості
- Мультимовність та мультимодальність

Продуктивність:

- Швидка генерація контенту
- Автоматизація рутинних задач
- Масштабованість рішень - 24/7 доступність

Якість:

- Людиноподібний текст
- Контекстне розуміння
- Креативність та різноманітність
- Постійне покращення

Доступність:

- API та хмарні сервіси
- Open-source альтернативи
- Зменшення вартості з часом
- Демократизація AI

Недоліки

Галюцинації:

- Генерація неправдивої інформації
- Впевненість у неправильних відповідях
- Складність верифікації
- Потенційна дезінформація

Обчислювальні вимоги:

- Величезні ресурси для навчання
- Значне енергоспоживання
- Дорогий inference для великих моделей
- Екологічний вплив

Етичні проблеми.

Обмеження:

- Відсутність справжнього розуміння
- Обмежений контекст (вікно уваги)
- Складність контролю виходу

- Залежність від якості промптів

Непередбачуваність:

- Недетерміністичні відповіді
- Складність відладки
- Чутливість до формулювань
- “Чорна скриня”

Контрольні запитання

1. Поясніть різницю між pre-training та fine-tuning в контексті LLM.
2. Що таке емерджентні властивості та при яких умовах вони виникають?
3. Як працює контекстне навчання та чому воно ефективне?
4. Опишіть основні етапи RLHF та їх призначення.
5. Яка роль токенізації в LLM та які виклики вона створює?
6. Поясніть концепцію KV-cache та її важливість для inference.
7. Які основні стратегії для роботи з обмеженим контекстним вікном?
8. Чим відрізняються різні стратегії вибірки (greedy, top-k, top-p)?
9. Як масштабування моделі впливає на її здібності?
10. Які основні підходи до зменшення галюцинацій в LLM?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження компактної мовної моделі

2. Мета

Зрозуміти принципи роботи LLM, реалізувати спрощену версію, дослідити вплив різних компонентів на якість генерації.

3. Завдання

- Реалізувати міні-LLM архітектуру
- Створити систему токенізації
- Імплементувати pre-training pipeline
- Додати різні decoding стратегії
- Реалізувати простий fine-tuning
- Дослідити вплив розміру моделі

4. Опис

Студенти створюють власну компактну LLM, що дозволяє експериментувати з різними аспектами архітектури та навчання.

5. Порядок виконання

1. Реалізувати BPE токенізатор

2. Створити decoder-only transformer

3. Імплементувати causal language modeling

4. Додати різні позиційні кодування

5. Реалізувати training loop з checkpointing

6. Створити inference pipeline

7. Додати метрики оцінки (perplexity)

8. Провести аблаційні експерименти

6. Результат

Функціональна міні-LLM з можливістю навчання на невеликих текстових корпусах та генерації тексту.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Клас LLMModel з повною архітектурою

- Tokenizer з BPE реалізацією

- Training pipeline з логуванням

- Inference engine з різними стратегіями

- Evaluation метрики

2. Продемонструвати роботу додатку:

- Pre-training на текстовому корпусі

- Генерація тексту з різними промптами

- Візуалізація learning curves

- Порівняння sampling стратегій

- Аналіз attention patterns

3. Бути готовим відповісти на питання:

- Як розмір моделі впливає на якість?

- Чому важливий warmup при навчанні?
- Як промпт впливає на генерацію?
- Які обмеження має реалізована модель?

Тести

1. Основна задача pre-training для LLM:
1) Класифікація тексту
2) Передбачення наступного токена
3) Машинний переклад
4) Аналіз тональності
2. RLHF використовується для:
1) Прискорення навчання
2) Зменшення розміру моделі
3) Вирівнювання з людськими преференціями
4) Покращення токенізації
3. Контекстне навчання означає:
1) Навчання на контексті
2) Точне налаштування на прикладах
3) Виконання задач через промпти без оновлення ваг
4) Контекстне вікно моделі
4. KV-cache використовується для:
1) Зберігання токенів
2) Прискорення генерації
3) Навчання моделі
4) Зменшення пам’яті
5. Температура при семплюванні контролює:
1) Швидкість генерації
2) Довжину відповіді
3) Випадковість/детермінованість виходу
4) Розмір контексту

Відповіді:

1-2,

2-3,

3-3,

4-2,

5-3

Тести підвищеної складності

1. Чому RMSNorm часто використовується замість LayerNorm в LLM?

1) Краща якість нормалізації

2) Менше обчислень через відсутність центрування

3) Більше параметрів

4) Необхідність для RoPE

2. SwiGLU активація в FFN відрізняється від ReLU:

1) Лінійністю

2) Гейтінг механізмом та гладкістю

3) Відсутністю нелінійності

4) Меншою кількістю параметрів

3. Мішана модель експертів (MoE) в LLM дозволяє:

1) Використовувати кілька моделей

2) Масштабувати модель з сублінійним зростанням обчислень

3) Покращити токенізацію

4) Зменшити контекстне вікно

4. Конституційний ШІ відрізняється від RLHF:

1) Відсутністю людського зворотного зв’язку

2) Використанням правил та самокритики

3) Швидшим навчанням

4) Меншою кількістю параметрів

5. Спекулятивне декодування прискорює інферування (inference) за рахунок:

1) Паралельну генерацію кількох токенів малою моделлю

2) Зменшення розміру моделі

3) Кешування всіх можливих продовжень

4) Використання GPU

6. Rope scaling дозволяє:

1) Збільшити розмір моделі

2) Екстраполювати на довші послідовності

3) Покращити якість

4) Зменшити пам’ять

7. Закони масштабування Chinchilla стверджують:

1) Більші моделі завжди кращі

2) Оптимальне співвідношення розміру моделі до даних

3) Необхідність RLHF

4) Важливість архітектури

8. BPE на рівні байтів відрізняється від BPE на рівні символів:

1) Кращою компресією

2) Можливістю кодування будь-якого тексту без unknown токенів

3) Меншим словником

4) Швидшою токенізацією

9. Градієнтний чекпоінтинг у великих мовних моделях (LLM):

1) Зберігає всі градієнти

2) Компроміс між пам’яттю та обчисленнями

3) Прискорює навчання

4) Покращує якість

10. DPO (пряме оптимізування переваг) порівняно з PPO:

1) Складніший в реалізації

2) Не потребує окремої моделі винагороди (reward model)

3) Дає гіршу якість

4) Вимагає більше даних

Відповіді:

1. 2 - RMSNorm ефективніший через відсутність операції центрування

2. 2 - SwiGLU використовує gate для модуляції активації

3. 2 - MoE активує підмножину експертів для кожного токена

4. 2 - Кноституційний AI використовує правила та самокритику

5. 1 - Мала модель генерує кандидатів, велика верифікує

6. 2 - Модифікація частот дозволяє роботу з довшими послідовностями

7. 2 - Обчислювально-оптимальне тренування (compute-optimal training) вимагає балансу моделі та даних

8. 2 - Рівень байтівгарантує можливість кодування будь-чого

9. 2 - Переобчислення активацій замість зберігання

10. 2 - DPO оптимізує переваги напряму без окремої моделі винагороди

Завдання для самостійної роботи

1. Реалізуйте різні стратегії позиційного кодування (learned, RoPE, ALiBi) та порівняйте їх.

2. Створіть систему для автоматизованого проєктування (інжинірингу) запитів (automatic prompt engineering) із використанням малої LLM.

3. Реалізуйте генерацію з підсиленням пошуком (retrieval-augmented generation, RAG) для вашої великої мовної моделі (LLM).

4. Дослідіть вплив різних схем зміни темпу навчання (learning rate schedules) на якість моделі..

5. Реалізуйте просту версію навчання з підкріпленням на основі людського зворотного зв’язку (RLHF) із синтетичними вподобаннями (synthetic preferences).

6. Створіть бенчмарк для оцінювання різних аспектів великих мовних моделей (LLM): знань, логічного мислення та творчості.

7. Реалізуйте злиття моделей (model merging) для комбінування кількох тонко налаштованих версій.

8. Розробіть систему для виявлення та зменшення галюцинацій.

9. Реалізуйте структуровану генерацію з гарантованим дотриманням формату (structured generation)..

10. Створіть інтерпретаційні інструменти для аналізу внутрішніх представлень LLM.

Теми лабораторних робіт

1. Побудова міні-LLM з нуля: від токенізації до генерації.

2. Стратегії точного налаштування: повний, LoRA, QLoRA порівняння.

3. Майстерня з інжинірингу (проєктування) запитів (prompt engineering): автоматизація та оптимізація.

4. RAG система: поєднання LLM з зовнішніми знаннями.

5. Мультимодальна LLM: розширення на зображення.

6. Ефективне інферування (efficient inference): квантизація та оптимізації.

7. Безпека та вирівнювання: реалізація базових захисних механізмів (guardrails).

8. Розподілене навчання: паралелізм моделі для великих мовних моделей (LLM).

9. Комплексний набір для оцінювання великих мовних моделей (evaluation suite).

10. Розробити застосунок: чат-бот з використанням LLM.

15.5. Зниження розмірності

У сучасному аналізі даних ми часто працюємо з багатовимірними векторами ознак — від сотень до тисяч змінних, що описують спостереження. Проте висока вимірність ускладнює не лише візуалізацію й інтерпретацію, а й ефективне навчання моделей: з'являється феномен «прокляття вимірності», коли дані стають надто розрідженими, а відстані між точками втрачають зміст.

Зниження розмірності — це набір методів, які перетворюють початковий набір змінних у простір нижчої розмірності, зберігаючи при цьому найбільш важливі структурні ознаки.

Зниження розмірності потрібне для:

- Візуалізації даних: зменшивши розмірність до 2–3 вимірів, ми можемо нанести спостереження на графік і побачити кластери, аномалії чи тренди.

- Шумопригнічення: видалення малозначущих компонент допомагає позбутися випадкового шуму й підвищити стійкість моделей.

- Прискорення обчислень: навчання на меншій кількості ознак знижує час і ресурси, потрібні для тренування алгоритмів.

- Запобігання перенавчанню: скорочення числа параметрів часто покращує узагальнювальну здатність моделі.

Потреба в такому процесі виникає коли:

- Коли кількість ознак значно перевищує розмір вибірки.

- Коли відстані в початковому просторі спотворюють реальні подібності.

- Коли важливо швидко візуалізувати або попередньо обробити дані з подальшим кластерним чи класифікаційним аналізом.

Зниження розмірності застосовують у задачах:

- Біоінформатики: аналіз генних експресій, де кожен зразок описується тисячами генів.

- Комп'ютерногой зору: попередня обробка ознак зображень перед класифікацією чи сегментацією.

- Обробки тексту: візуалізація векторних уявлень слів (word embeddings) або документів.

- Маркетингу і рекомендаційних системах: виявлення сегментів користувачів на основі багатовимірних поведінкових характеристик.

У цьому розділі спочатку розглянемо аналіз головних компонент (PCA) — лінійний метод, який шукає ортогональні напрямки з максимальною дисперсією даних, пояснюючи більшу частину варіації мінімальною кількістю компонент. Після цього познайомимося з t-SNE — нелінійним алгоритмом, що зберігає локальні відносини між об'єктами за допомогою стохастичного вбудовування ймовірностей, дозволяючи виявити чіткі кластери навіть у складних розподілах. Далі ми детально розберемо їхні математичні основи, практичні налаштування та конкретні приклади застосування.

Метод головних компонент (PCA)

Теоретичні відомості

Метод головних компонент (Principal Component Analysis, PCA) є одним з найпоширеніших та найефективніших методів зниження розмірності даних. Цей статистичний метод був розроблений Карлом Пірсоном у 1901 році та незалежно сформульований Гарольдом Хотеллінгом у 1933 році. PCA перетворює набір можливо корельованих змінних у набір лінійно некорельованих змінних, які називаються головними компонентами. Основна ідея методу полягає у знаходженні нових осей координат (головних компонент), вздовж яких дані мають максимальну дисперсію.

Нехай маємо набір даних X, що складається з n спостережень та p ознак. Кожне спостереження можна представити як вектор $x_i \in \mathbb{R}^p$, де $i = 1,2,\ldots,n$. Матрицю даних можна записати як:

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix}$$

Першим кроком алгоритму PCA є центрування даних. Для кожної ознаки обчислюється середнє значення:

$$\mu_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$$

де $j = 1,2,\ldots,p$. Центровані дані отримуються шляхом віднімання середнього значення від кожного елемента:

$$\tilde{x}_{ij} = x_{ij} - \mu_j$$

Після центрування обчислюється коваріаційна матриця:

$$C = \frac{1}{n-1}\tilde{X}^T\tilde{X}$$

де $\tilde{X}$ - матриця центрованих даних. Коваріаційна матриця C є симетричною матрицею розміру $p \times p$, елементи якої представляють коваріації між парами ознак.

Власні значення та власні вектори

Головні компоненти визначаються через власні вектори коваріаційної матриці. Для знаходження власних векторів та власних значень розв'язується характеристичне рівняння:

$$Cv = \lambda v$$

де v - власний вектор, а λ - відповідне власне значення. Це рівняння можна переписати як:

$$(C - \lambda I)v = 0$$

де I - одинична матриця. Для існування нетривіального розв'язку необхідно, щоб:

$$\det(C - \lambda I) = 0$$

Розв'язання цього рівняння дає p власних значень $\lambda_1, \lambda_2, \ldots, \lambda_p$, які зазвичай впорядковуються за спаданням: $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p \geq 0$.

Геометрична інтерпретація

З геометричної точки зору, PCA знаходить нові ортогональні осі в просторі даних, такі що:

- Перша головна компонента відповідає напрямку максимальної дисперсії даних

- Друга головна компонента ортогональна до першої та відповідає напрямку максимальної дисперсії серед залишкових напрямків

- Кожна наступна компонента ортогональна до всіх попередніх та максимізує дисперсію в залишковому підпросторі

Проекція даних

Після знаходження власних векторів, дані проектуються на новий базис, утворений головними компонентами. Нехай W - матриця, стовпцями якої є власні вектори, відповідні k найбільшим власним значенням. Тоді проекція даних на k-вимірний підпростір обчислюється як:

$$Y = \tilde{X}W$$

де Y - матриця розміру $n \times k$, що містить координати спостережень у новому базисі.

Вибір кількості компонент

Важливим питанням є визначення кількості головних компонент, які слід зберегти. Існує кілька критеріїв:

Критерій збереженої дисперсії: зберігаються компоненти, що пояснюють задану частку загальної дисперсії. Частка поясненої дисперсії для перших k компонент:

$$\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i}$$

Критерій Кайзера: зберігаються компоненти з власними значеннями більшими за 1 (для стандартизованих даних).

Метод ліктя: будується графік власних значень або поясненої дисперсії і шукається точка “зламу”.

Стандартизація даних

Важливо зазначити, що PCA чутливий до масштабу ознак. Якщо ознаки мають різні одиниці вимірювання або суттєво відрізняються за величиною, необхідно провести стандартизацію:

$$z_{ij} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

де σ_j - стандартне відхилення j-ї ознаки.

PCA тісно пов'язаний з сингулярним розкладом (SVD) матриці даних. Якщо провести SVD центрованої матриці даних:

$$\tilde{X} = U\Sigma V^T$$

то стовпці матриці V є власними векторами коваріаційної матриці, а діагональні елементи $\Sigma^2/(n-1)$ відповідають власним значенням.

Псевдокод алгоритму PCA

```
функція PCA(X, k)
    // X - матриця даних розміру n × p
    // k - кількість головних компонент для збереження

    n = кількість_рядків(X)
    p = кількість_стовпців(X)

    // Крок 1: Обчислення середніх значень
    середні = вектор(p)
    для j = 1 до p виконати
        середні[j] = 0
        для i = 1 до n виконати
            середні[j] = середні[j] + X[i,j]
        середні[j] = середні[j] / n

    // Крок 2: Центрування даних
    X_центровані = матриця(n, p)
    для i = 1 до n виконати
        для j = 1 до p виконати
            X_центровані[i,j] = X[i,j] - середні[j]

    // Крок 3: Обчислення коваріаційної матриці
    C = матриця(p, p)
    для i = 1 до p виконати
        для j = 1 до p виконати
            C[i,j] = 0
            для m = 1 до n виконати
                C[i,j] = C[i,j] + X_центровані[m,i] * X_центровані[m,
j]
            C[i,j] = C[i,j] / (n - 1)

    // Крок 4: Знаходження власних значень та векторів
```

```
    (власні_значення, власні_вектори) = ВласніЗначенняТаВектори(C)

    // Крок 5: Сортування власних значень за спаданням
    індекси = СортуватиІндексиЗаСпаданням(власні_значення)

    // Крок 6: Вибір k найбільших власних векторів
    W = матриця(p, k)
    для j = 1 до k виконати
        для i = 1 до p виконати
            W[i,j] = власні_вектори[i, індекси[j]]

    // Крок 7: Проекція даних
    Y = матриця(n, k)
    для i = 1 до n виконати
        для j = 1 до k виконати
            Y[i,j] = 0
            для m = 1 до p виконати
                Y[i,j] = Y[i,j] + X_центровані[i,m] * W[m,j]

    повернути Y, W, власні_значення, середні

функція ВласніЗначенняТаВектори(A)
    // Метод степеневої ітерації для знаходження власних значень
    // Для повної реалізації використовується QR-алгоритм

    p = розмір(A)
    власні_вектори = одинична_матриця(p)
    власні_значення = вектор(p)

    // QR-ітерації
    A_копія = копія(A)
    для ітерація = 1 до 1000 виконати
        (Q, R) = QR_розклад(A_копія)
        A_копія = R * Q
        власні_вектори = власні_вектори * Q

        // Перевірка збіжності
        якщо максимум_позадіагональних_елементів(A_копія) < 1e-10 тод
і
            перервати

    // Витягнути діагональні елементи як власні значення
    для i = 1 до p виконати
        власні_значення[i] = A_копія[i,i]
```

```
    повернути (власні_значення, власні_вектори)

функція ВідновитиДані(Y, W, середні)
    // Відновлення даних з головних компонент

    n = кількість_рядків(Y)
    k = кількість_стовпців(Y)
    p = кількість_рядків(W)

    X_відновлені = матриця(n, p)

    для i = 1 до n виконати
        для j = 1 до p виконати
            X_відновлені[i,j] = середні[j]
            для m = 1 до k виконати
                X_відновлені[i,j] = X_відновлені[i,j] + Y[i,m] * W[j,
m]

    повернути X_відновлені

функція ОбчислитиПоясненуДисперсію(власні_значення, k)
    // Обчислення частки поясненої дисперсії

    загальна_дисперсія = 0
    пояснена_дисперсія = 0

    для i = 1 до довжина(власні_значення) виконати
        загальна_дисперсія = загальна_дисперсія + власні_значення[i]
        якщо i <= k тоді
            пояснена_дисперсія = пояснена_дисперсія + власні_значення
[i]

    частка = пояснена_дисперсія / загальна_дисперсія
    повернути частка

функція СтандартизуватиДані(X)
    // Стандартизація даних

    n = кількість_рядків(X)
    p = кількість_стовпців(X)

    X_стандартизовані = матриця(n, p)

    для j = 1 до p виконати
        середнє = 0
```

```
        для i = 1 до n виконати
            середнє = середнє + X[i,j]
        середнє = середнє / n

        дисперсія = 0
        для i = 1 до n виконати
            дисперсія = дисперсія + (X[i,j] - середнє)^2
        дисперсія = дисперсія / (n - 1)
        стд_відхилення = корінь(дисперсія)

        для i = 1 до n виконати
            якщо стд_відхилення > 0 тоді
                X_стандартизовані[i,j] = (X[i,j] - середнє) / стд_від
хилення
            інакше
                X_стандартизовані[i,j] = 0

    повернути X_стандартизовані
```

Приклад застосування

Розглянемо простий приклад застосування PCA для двовимірних даних. Нехай маємо 5 точок на площині:

```
Точка 1: (2.5, 2.4)
Точка 2: (0.5, 0.7)
Точка 3: (2.2, 2.9)
Точка 4: (1.9, 2.2)
Точка 5: (3.1, 3.0)
```

Крок 1: Обчислення середніх значень

Середнє для x: $\mu_x = \frac{2.5+0.5+2.2+1.9+3.1}{5} = \frac{10.2}{5} = 2.04$

Середнє для y: $\mu_y = \frac{2.4+0.7+2.9+2.2+3.0}{5} = \frac{11.2}{5} = 2.24$

Крок 2: Центрування даних

```
Центровані дані:
Точка 1: (0.46, 0.16)
Точка 2: (-1.54, -1.54)
Точка 3: (0.16, 0.66)
Точка 4: (-0.14, -0.04)
Точка 5: (1.06, 0.76)
```

Крок 3: Обчислення коваріаційної матриці

$$C = \frac{1}{4}\begin{pmatrix} \sum(x_i - \mu_x)^2 & \sum(x_i - \mu_x)(y_i - \mu_y) \\ \sum(x_i - \mu_x)(y_i - \mu_y) & \sum(y_i - \mu_y)^2 \end{pmatrix}$$

Після обчислень отримуємо:

$$C = \begin{pmatrix} 0.8165 & 0.7365 \\ 0.7365 & 0.7165 \end{pmatrix}$$

Крок 4: Знаходження власних значень

Розв'язуємо характеристичне рівняння:

$$\det(C - \lambda I) = 0$$

$$(0.8165 - \lambda)(0.7165 - \lambda) - 0.7365^2 = 0$$

Після розв'язання отримуємо:

- $\lambda_1 = 1.454$ (перше власне значення)

- $\lambda_2 = 0.079$ (друге власне значення)

Крок 5: Знаходження власних векторів

Для $\lambda_1 = 1.454$:

$$v_1 = \begin{pmatrix} 0.735 \\ 0.678 \end{pmatrix}$$

Для $\lambda_2 = 0.079$:

$$v_2 = \begin{pmatrix} -0.678 \\ 0.735 \end{pmatrix}$$

Крок 6: Проекція даних на першу головну компоненту

Проекція кожної точки обчислюється як скалярний добуток центрованої точки на власний вектор:

```
Точка 1: 0.46 × 0.735 + 0.16 × 0.678 = 0.447
Точка 2: -1.54 × 0.735 + (-1.54) × 0.678 = -2.176
Точка 3: 0.16 × 0.735 + 0.66 × 0.678 = 0.565
Точка 4: -0.14 × 0.735 + (-0.04) × 0.678 = -0.130
Точка 5: 1.06 × 0.735 + 0.76 × 0.678 = 1.294
```

Аналіз результатів

Перша головна компонента пояснює:

$$\frac{\lambda_1}{\lambda_1 + \lambda_2} = \frac{1.454}{1.454 + 0.079} = 0.948 = 94.8\%$$

дисперсії даних. Це означає, що можна зменшити розмірність з 2 до 1, втративши лише 5.2% інформації.

Область застосування

PCA широко використовується в різних галузях науки та техніки:

Обробка зображень та комп'ютерний зір

- Розпізнавання облич (метод Eigenfaces)
- Стиснення зображень
- Видалення шуму
- Виявлення аномалій у відеопотоці

Біоінформатика та медицина

- Аналіз генетичних даних
- Візуалізація багатовимірних медичних даних
- Класифікація захворювань
- Аналіз медичних зображень

Фінанси та економіка

- Управління портфелем активів
- Аналіз ринкових ризиків
- Виявлення факторів, що впливають на економічні показники
- Прогнозування фінансових часових рядів

Обробка сигналів

- Фільтрація шуму
- Стиснення аудіо та відео
- Розпізнавання мовлення
- Аналіз спектральних даних

Машинне навчання

- Попередня обробка даних
- Візуалізація багатовимірних даних
- Регуляризація моделей
- Прискорення навчання алгоритмів

Приклади застосування

Приклад 1: Розпізнавання облич методом Eigenfaces

У системі розпізнавання облич кожне зображення розміром 100×100 пікселів представляється як вектор з 10000 елементів. База даних містить 1000 зображень 100 різних людей.

Приклад 2: Аналіз фінансового ринку

Інвестиційна компанія аналізує 50 акцій протягом 5 років, маючи щоденні дані про ціни. PCA застосовується для:

- Виявлення основних факторів, що впливають на рух ринку

Переваги та недоліки

Переваги

Ефективне зниження розмірності

- Оптимальне збереження дисперсії даних
- Математично обґрунтований підхід
- Можливість вибору необхідної кількості компонент

Обчислювальна ефективність

- Існують швидкі алгоритми для великих даних
- Можливість інкрементального навчання
- Паралелізація обчислень

Недоліки

Лінійність перетворення

- Не може виявляти нелінійні залежності
- Обмежена ефективність для складних структур даних
- Необхідність нелінійних розширень (Kernel PCA)

Чутливість до масштабу

- Необхідність попередньої стандартизації
- Домінування ознак з великою дисперсією
- Можлива втрата важливої інформації

Складність інтерпретації

- Головні компоненти є лінійними комбінаціями всіх ознак
- Втрата фізичного змісту
- Труднощі в поясненні результатів

Припущення про нормальність

- Оптимальність для гаусівських розподілів
- Можлива неефективність для інших розподілів
- Чутливість до викидів

Контрольні запитання

1. Що таке головні компоненти та як вони визначаються математично?

2. Чому необхідно центрувати дані перед застосуванням PCA?

3. Яка різниця між коваріаційною та кореляційною матрицями в контексті PCA?

4. Як вибрати оптимальну кількість головних компонент для конкретної задачі?

5. Що означає власне значення головної компоненти з точки зору дисперсії?

6. Чому власні вектори коваріаційної матриці ортогональні?

7. Як PCA пов’язаний з сингулярним розкладом матриці?

8. Які існують альтернативи PCA для нелінійного зниження розмірності?

9. Як впливає наявність викидів на результати PCA?

10. В яких випадках PCA може бути неефективним?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження методу головних компонент

2. Мета

Вивчити теоретичні основи методу PCA, реалізувати алгоритм, дослідити його властивості та застосувати для аналізу реальних даних.

3. Завдання

- Реалізувати алгоритм PCA з нуля

- Порівняти результати з бібліотечними реалізаціями

- Візуалізувати головні компоненти

- Дослідити вплив стандартизації на результати

- Застосувати PCA для стиснення зображень

- Проаналізувати залежність якості відновлення від кількості компонент

4. Опис

Лабораторна робота передбачає поетапну реалізацію алгоритму PCA, починаючи з простих двовимірних даних та закінчуючи застосуванням до реальних багатовимірних датасетів. Студенти повинні реалізувати всі

кроки алгоритму, включаючи обчислення власних значень та векторів, проекцію даних та їх відновлення.

5. Порядок виконання

1. Генерація тестових двовимірних даних з відомою структурою

2. Реалізація функцій центрування та стандартизації

3. Обчислення коваріаційної матриці

4. Реалізація методу знаходження власних значень та векторів

5. Проекція даних на головні компоненти

6. Візуалізація результатів для 2D та 3D даних

7. Застосування до датасету Iris

8. Реалізація стиснення зображень за допомогою PCA

9. Аналіз похибки відновлення

10. Порівняння з sklearn.decomposition.PCA

6. Результат

Програмна реалізація PCA з можливістю візуалізації результатів, графіки залежності поясненої дисперсії від кількості компонент, демонстрація стиснення зображень з різними рівнями компресії.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:

- Функція центрування даних

- Функція обчислення коваріаційної матриці

- Функція знаходження власних значень та векторів

- Функція проекції даних на головні компоненти

- Функція відновлення даних

- Функція візуалізації результатів

2. Продемонструвати роботу додатку:

- Візуалізація головних компонент для 2D даних

- Графік scree plot (власні значення)

- Графік накопиченої поясненої дисперсії

- Результати стиснення зображення з різними рівнями

- Порівняння часу виконання з бібліотечною реалізацією

3. Бути готовим відповісти на питання:

- Як змінюються результати при використанні коваріаційної та кореляційної матриць?

- Який вплив має порядок сортування власних значень?

- Як вибрати оптимальну кількість компонент для конкретної задачі?

- Які обмеження має лінійний PCA?

Тести

1. Головні компоненти в PCA є:

1) Початковими ознаками даних

2) Власними векторами коваріаційної матриці

3) Середніми значеннями ознак

4) Випадковими векторами

2. Перша головна компонента відповідає напрямку:

1) Мінімальної дисперсії

2) Максимальної дисперсії

3) Нульової дисперсії

4) Середньої дисперсії

3. Для стандартизованих даних сума всіх власних значень дорівнює:

1) Нулю

2) Одиниці

3) Кількості ознак

4) Кількості спостережень

4. PCA зберігає:

1) Евклідові відстані між точками

2) Кути між векторами

3) Максимальну дисперсію при проекції

4) Початкові значення ознак

5. Якщо дві ознаки повністю корельовані, то:

1) PCA не працює

2) Одне з власних значень дорівнює нулю

3) Всі власні значення однакові

4) Потрібна додаткова нормалізація

Відповіді:

1-2,

2-2,

3-3,

4-3,

5-2

Тести підвищеної складності

1. При застосуванні PCA до матриці n×p, де n < p, максимальна кількість ненульових власних значень:

1) p

2) n

3) n-1

4) p-1

2. Kernel PCA відрізняється від звичайного PCA тим, що:

1) Використовує іншу метрику відстані

2) Виконує неявне нелінійне перетворення через ядрові функції

3) Не потребує центрування даних

4) Працює тільки з категоріальними даними

3. Якщо застосувати PCA двічі поспіль до одних даних, то:

1) Результат покращиться

2) Результат не зміниться

3) Дисперсія збільшиться

4) З’являться нові компоненти

4. Robust PCA відрізняється від класичного PCA:

1) Швидкістю обчислень

2) Стійкістю до викидів через використання L1-норми

3) Кількістю компонент

4) Необхідністю стандартизації

5. При інкрементальному PCA:

1) Потрібно зберігати всі дані в пам’яті

2) Можна оновлювати модель при надходженні нових даних

3) Результат гірший за batch PCA

4) Неможливо обчислити власні значення

6. Sparse PCA накладає обмеження:

1) На кількість ненульових елементів у власних векторах

2) На величину власних значень

3) На розмір даних

4) На тип даних

7. Якщо дані лежать на нелінійному многовиді малої розмірності:

1) PCA знайде цей многовид

2) PCA може дати неоптимальне представлення

3) PCA не буде працювати

4) Потрібна додаткова нормалізація

8. Зв’язок між PCA та факторним аналізом:

1) Це одне й те саме

2) PCA моделює загальну дисперсію, факторний аналіз - спільну

3) Факторний аналіз є окремим випадком PCA

4) Немає зв’язку

9. При whitening (вибілюванні) даних після PCA:

1) Дані стають некорельованими

2) Дані стають некорельованими з одиничною дисперсією

3) Зберігається більше інформації

4) Покращується інтерпретованість

10. Probabilistic PCA відрізняється від класичного:

1) Використанням ймовірнісної моделі з латентними змінними

2) Кращою точністю

3) Меншою кількістю параметрів

4) Відсутністю власних векторів

Відповіді:

1. 3 - Ранг центрованої матриці не перевищує min(n-1, p)

2. 2 - Kernel PCA використовує kernel trick для нелінійних перетворень

3. 2 - PCA є ідемпотентною операцією

4. 2 - Robust PCA мінімізує L1-норму для стійкості до викидів

5. 2 - Incremental PCA дозволяє онлайн оновлення

6. 1 - Sparse PCA обмежує кількість ненульових коефіцієнтів

7. 2 - Лінійний PCA не оптимальний для нелінійних структур

8. 2 - Різні моделі дисперсії: загальна vs спільна

9. 2 - Whitening створює сферичний розподіл даних

10. 1 - PPCA має ймовірнісну інтерпретацію через латентні змінні

Завдання для самостійної роботи

1. Реалізуйте PCA з використанням методу степеневих ітерацій для знаходження власних векторів.

2. Порівняйте ефективність PCA та автоенкодерів для стиснення зображень різних типів.

3. Дослідіть вплив різних метрик відстані на результати PCA.

4. Реалізуйте Kernel PCA з різними ядровими функціями та порівняйте з лінійним PCA.

5. Створіть інтерактивну візуалізацію для демонстрації роботи PCA на 3D даних.

6. Застосуйте PCA для аналізу часових рядів (наприклад, фінансових даних).

7. Реалізуйте Incremental PCA для обробки даних, що не поміщаються в пам’ять.

8. Дослідіть використання PCA для виявлення аномалій у багатовимірних даних.

9. Порівняйте PCA з іншими методами зниження розмірності (t-SNE, UMAP).

10. Створіть веб-додаток для візуалізації та аналізу даних за допомогою PCA.

Теми лабораторних робіт

1. Реалізація базового алгоритму PCA та його застосування до датасету Iris.

2. Використання PCA для розпізнавання рукописних цифр (датасет MNIST).

3. Стиснення кольорових зображень за допомогою PCA: аналіз компромісу між якістю та розміром.

4. Застосування PCA для аналізу багатоспектральних супутникових знімків.

5. Порівняльний аналіз лінійного PCA та Kernel PCA на нелінійних даних.

6. Використання PCA для візуалізації багатовимірних даних у біоінформатиці.

7. Реалізація Robust PCA для обробки зашумлених даних з викидами.

8. Застосування PCA в рекомендаційних системах для зниження розмірності матриці рейтингів.

9. Дослідження Sparse PCA для інтерпретованого зниження розмірності.

10. Розробка системи моніторингу промислових процесів на основі PCA.

Зниження розмірності: t-SNE

Теоретичні відомості

Метод t-SNE (t-distributed Stochastic Neighbor Embedding) є потужним алгоритмом нелінійного зниження розмірності, розробленим Лоренсом ван дер Маатеном та Джеффрі Хінтоном у 2008 році. Цей метод особливо ефективний для візуалізації багатовимірних даних у двовимірному або тривимірному просторі, зберігаючи локальну структуру даних та виявляючи кластери на різних масштабах.

Алгоритм t-SNE базується на ідеї збереження подібності між точками при переході з високовимірного простору в низьковимірний. Замість збереження відстаней, як це роблять лінійні методи типу PCA, t-SNE зберігає ймовірнісні розподіли, що описують схожість між парами точок.

Процес складається з двох основних етапів:

1. Обчислення ймовірностей схожості між парами точок у високовимірному просторі

2. Знаходження відповідного низьковимірного представлення, що мінімізує розбіжність між ймовірнісними розподілами

Нехай маємо набір даних $X = \{x_1, x_2, \ldots, x_n\}$ у високовимірному просторі $\mathbb{R}^d$, де d - розмірність вихідних даних. Метою є знаходження низьковимірного представлення $Y = \{y_1, y_2, \ldots, y_n\}$ у просторі $\mathbb{R}^m$, де зазвичай $m = 2$ або $m = 3$.

Крок 1: Обчислення подібностей у високовимірному просторі

Спочатку обчислюються умовні ймовірності $p_{j|i}$, що точка x_j є сусідом точки x_i:

$$p_{j|i} = \frac{\exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2/2\sigma_i^2)}$$

де σ_i - ширина гаусового ядра для точки x_i. Для $i = j$ приймається $p_{i|i} = 0$.

Параметр σ_i вибирається адаптивно для кожної точки так, щоб ефективна кількість сусідів (perplexity) була приблизно однаковою. Perplexity визначається як:

$$\text{Perp}(P_i) = 2^{H(P_i)}$$

де $H(P_i)$ - ентропія Шеннона розподілу P_i:

$$H(P_i) = -\sum_j p_{j|i} \log_2 p_{j|i}$$

Для симетризації обчислюються спільні ймовірності:

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}$$

Крок 2: Обчислення подібностей у низьковимірному просторі

У низьковимірному просторі використовується t-розподіл Стьюдента з одним ступенем свободи (розподіл Коші) замість гаусового розподілу:

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neq l} (1 + ||y_k - y_l||^2)^{-1}}$$

Використання t-розподілу дозволяє краще моделювати великі відстані та запобігає проблемі “скупчення” точок у центрі візуалізації.

Крок 3: Мінімізація розбіжності Кульбака-Лейблера

Оптимальне низьковимірне представлення знаходиться шляхом мінімізації розбіжності Кульбака-Лейблера між розподілами P та Q:

$$KL(P||Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

Градієнт цільової функції по координатах точки y_i:

$$\frac{\partial KL}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}$$

Оптимізація

Для мінімізації функції втрат використовується градієнтний спуск з додатковими техніками:

Momentum: для прискорення збіжності та уникнення локальних мінімумів:

$$Y^{(t)} = Y^{(t-1)} + \eta\frac{\partial KL}{\partial Y} + \alpha(t)(Y^{(t-1)} - Y^{(t-2)})$$

де η - швидкість навчання, $\alpha(t)$ - коефіцієнт momentum.

Early exaggeration: на початкових ітераціях множать p_{ij} на константу (зазвичай 4 або 12), що допомагає формуванню чітких кластерів.

Адаптивна швидкість навчання: автоматичне налаштування швидкості навчання для кожної координати.

Обчислювальна складність

Наївна реалізація t-SNE має квадратичну складність $O(n^2)$ як по часу, так і по пам'яті, що робить її непрактичною для великих датасетів. Існують наближені методи:

Barnes-Hut t-SNE: використовує дерева для апроксимації далеких взаємодій, зменшуючи складність до $O(n\log n)$.

FIt-SNE: використовує швидке перетворення Фур'є для ще швидших обчислень.

Вибір параметрів

Perplexity: ключовий параметр, що визначає баланс між локальною та глобальною структурою. Типові значення: 5-50. Більші значення враховують більше сусідів.

Кількість ітерацій: зазвичай 1000-5000 ітерацій достатньо для збіжності.

Швидкість навчання: типові значення 10-1000, залежно від розміру датасету.

Псевдокод алгоритму t-SNE

```
функція tSNE(X, розмірність_виходу, perplexity, кількість_ітерацій, ш
видкість_навчання)
    // X - матриця даних n × d
    // розмірність_виходу - зазвичай 2 або 3
    // perplexity - ефективна кількість сусідів (5-50)
```

```
n = кількість_рядків(X)

// Крок 1: Обчислення попарних відстаней
D = матриця(n, n)
для i = 1 до n виконати
    для j = 1 до n виконати
        D[i,j] = евклідова_відстань(X[i], X[j])

// Крок 2: Обчислення матриці ймовірностей P
P = ОбчислитиЙмовірності(D, perplexity)

// Крок 3: Ініціалізація низьковимірного представлення
Y = випадкова_матриця(n, розмірність_виходу) * 0.0001

// Крок 4: Ініціалізація параметрів оптимізації
dY = матриця_нулів(n, розмірність_виходу)
iY = матриця_нулів(n, розмірність_виходу)
gains = матриця_одиниць(n, розмірність_виходу)

// Крок 5: Оптимізація
для ітерація = 1 до кількість_ітерацій виконати
    // Early exaggeration для перших 250 ітерацій
    якщо ітерація <= 250 тоді
        P_ефективне = P * 4
    інакше
        P_ефективне = P

    // Обчислення матриці Q
    Q = ОбчислитиНизьковимірніЙмовірності(Y)

    // Обчислення градієнта
    dY = ОбчислитиГрадієнт(P_ефективне, Q, Y)

    // Адаптивна швидкість навчання
    для i = 1 до n виконати
        для j = 1 до розмірність_виходу виконати
            якщо знак(dY[i,j]) != знак(iY[i,j]) тоді
                gains[i,j] = gains[i,j] + 0.2
            інакше
                gains[i,j] = gains[i,j] * 0.8
            якщо gains[i,j] < 0.01 тоді
                gains[i,j] = 0.01

    // Оновлення з momentum
```

```
        якщо ітерація < 250 тоді
            momentum = 0.5
        інакше
            momentum = 0.8

        iY = momentum * iY - швидкість_навчання * (gains * dY)
        Y = Y + iY

        // Центрування
        Y = Y - середнє(Y)

        // Вивід прогресу
        якщо ітерація % 100 == 0 тоді
            C = ОбчислитиВартість(P_ефективне, Q)
            вивести "Ітерація", ітерація, ": вартість =", C

    повернути Y

функція ОбчислитиЙмовірності(D, perplexity)
    n = розмір(D, 1)
    P = матриця(n, n)
    бета = вектор_одиниць(n)
    логU = логарифм(perplexity)

    // Для кожної точки знайти відповідну сигму
    для i = 1 до n виконати
        // Бінарний пошук для сигми
        бета_мін = -нескінченність
        бета_макс = нескінченність

        для спроба = 1 до 50 виконати
            // Обчислення умовних ймовірностей
            сума = 0
            для j = 1 до n виконати
                якщо i != j тоді
                    P[i,j] = експонента(-D[i,j]^2 * бета[i])
                    сума = сума + P[i,j]

            // Нормалізація
            для j = 1 до n виконати
                якщо i != j тоді
                    P[i,j] = P[i,j] / сума
                інакше
                    P[i,j] = 0
```

```
            // Обчислення ентропії
            H = 0
            для j = 1 до n виконати
                якщо P[i,j] > 0 тоді
                    H = H - P[i,j] * логарифм(P[i,j])

            // Оновлення бета
            різниця = H - логU
            якщо абс(різниця) < 1e-5 тоді
                перервати

            якщо різниця > 0 тоді
                бета_мін = бета[i]
                якщо бета_макс == нескінченність тоді
                    бета[i] = бета[i] * 2
                інакше
                    бета[i] = (бета[i] + бета_макс) / 2
            інакше
                бета_макс = бета[i]
                якщо бета_мін == -нескінченність тоді
                    бета[i] = бета[i] / 2
                інакше
                    бета[i] = (бета[i] + бета_мін) / 2

    // Симетризація
    для i = 1 до n виконати
        для j = i+1 до n виконати
            P[i,j] = (P[i,j] + P[j,i]) / (2 * n)
            P[j,i] = P[i,j]

    повернути P

функція ОбчислитиНизьковимірніЙмовірності(Y)
    n = кількість_рядків(Y)
    Q = матриця(n, n)

    // Обчислення чисельників
    сума = 0
    для i = 1 до n виконати
        для j = 1 до n виконати
            якщо i != j тоді
                відстань_квадрат = сума_квадратів(Y[i] - Y[j])
                Q[i,j] = 1 / (1 + відстань_квадрат)
                сума = сума + Q[i,j]
```

```
    // Нормалізація
    для i = 1 до n виконати
        для j = 1 до n виконати
            Q[i,j] = максимум(Q[i,j] / сума, 1e-12)

    повернути Q

функція ОбчислитиГрадієнт(P, Q, Y)
    n = кількість_рядків(Y)
    d = кількість_стовпців(Y)
    dY = матриця_нулів(n, d)

    для i = 1 до n виконати
        для j = 1 до n виконати
            якщо i != j тоді
                множник = 4 * (P[i,j] - Q[i,j])
                відстань_квадрат = сума_квадратів(Y[i] - Y[j])
                коефіцієнт = множник / (1 + відстань_квадрат)
                для k = 1 до d виконати
                    dY[i,k] = dY[i,k] + коефіцієнт * (Y[i,k] - Y[j,k]
)

    повернути dY

функція ОбчислитиВартість(P, Q)
    // Обчислення KL-дивергенції
    вартість = 0
    n = кількість_рядків(P)

    для i = 1 до n виконати
        для j = 1 до n виконати
            якщо P[i,j] > 0 і Q[i,j] > 0 тоді
                вартість = вартість + P[i,j] * логарифм(P[i,j] / Q[i,
j])

    повернути вартість
```

Модифікації та розширення

1. Parametric t-SNE: використовує нейронну мережу для відображення, що дозволяє проектувати нові точки.

2. Multiple maps t-SNE: створює кілька різних візуалізацій для виявлення різних аспектів даних.

3. Heavy-tailed t-SNE: використовує розподіли з важчими хвостами для кращого розділення кластерів.

Приклад застосування

Розглянемо застосування t-SNE до простого набору даних з трьома кластерами у тривимірному просторі.

Вхідні дані: 15 точок у 3D просторі, що утворюють три кластери:

Кластер 1 (червоний):

```
(1.0, 1.0, 1.0)
(1.2, 0.8, 1.1)
(0.9, 1.1, 0.9)
(1.1, 0.9, 1.2)
(0.8, 1.2, 1.0)
```

Кластер 2 (зелений):

```
(5.0, 5.0, 1.0)
(5.2, 4.8, 1.1)
(4.9, 5.1, 0.9)
(5.1, 4.9, 1.2)
(4.8, 5.2, 1.0)
```

Кластер 3 (синій):

```
(1.0, 5.0, 5.0)
(1.2, 4.8, 5.1)
(0.9, 5.1, 4.9)
(1.1, 4.9, 5.2)
(0.8, 5.2, 5.0)
```

Крок 1: Обчислення матриці відстаней

Обчислюємо евклідові відстані між всіма парами точок. Наприклад, відстань між першими точками кластерів 1 та 2:

$$d_{1,6} = \sqrt{(1-5)^2 + (1-5)^2 + (1-1)^2} = \sqrt{16 + 16 + 0} = 5.66$$

Крок 2: Обчислення ймовірностей у високовимірному просторі

Для perplexity = 5, знаходимо відповідні значення σ_i для кожної точки. Наприклад, для першої точки:

- Сусіди в межах кластера мають високі ймовірності (≈0.2-0.3)

- Точки з інших кластерів мають низькі ймовірності (≈0.001-0.01)

Крок 3: Ініціалізація та оптимізація

Починаємо з випадкового розміщення точок у 2D:

```
Y початкове (випадкове):
(0.001, -0.002)
```

```
(-0.001, 0.003)
...
```
Ітерація 100: Точки починають групуватися:

```
Кластер 1: навколо (-2, -1)
Кластер 2: навколо (3, 0)
Кластер 3: навколо (-1, 3)
```
Ітерація 500: Чіткі кластери сформовані:

```
Кластер 1: щільна група навколо (-5, -3)
Кластер 2: щільна група навколо (6, 1)
Кластер 3: щільна група навколо (-2, 5)
```
Фінальний результат (ітерація 1000):

Три чітко розділені кластери у 2D просторі, де:

- Відстані між кластерами приблизно пропорційні оригінальним

- Точки всередині кластерів залишаються близькими

- Загальна структура даних збережена

Аналіз результату:

KL-дивергенція зменшилася з початкових ~4.5 до фінальних ~0.8, що свідчить про хорошу збіжність. Візуалізація чітко показує три окремі групи, що відповідають оригінальним кластерам у 3D просторі.

Область застосування

t-SNE знайшов широке застосування в багатьох галузях завдяки своїй здатності виявляти складні нелінійні структури:

Біоінформатика та геноміка

- Візуалізація даних одноклітинного секвенування (scRNA-seq)
- Аналіз експресії генів
- Кластеризація клітинних популяцій
- Виявлення підтипів захворювань

Комп’ютерний зір

- Візуалізація простору ознак нейронних мереж
- Аналіз представлень зображень
- Дослідження learned embeddings
- Виявлення аномалій у зображеннях

Обробка природної мови

- Візуалізація word embeddings
- Аналіз семантичної подібності текстів
- Дослідження мовних моделей
- Кластеризація документів

Машинне навчання

- Розвідувальний аналіз даних
- Візуалізація високовимірних датасетів
- Оцінка якості ознак
- Діагностика моделей

Нейронаука

- Аналіз нейронної активності
- Візуалізація патернів мозкової діяльності
- Кластеризація нейронів за функціями

Соціальні науки

- Аналіз опитувань
- Виявлення груп у соціальних мережах
- Візуалізація поведінкових патернів

Приклади застосування

Приклад 1: Аналіз одноклітинного RNA секвенування

Дослідницька лабораторія використовує t-SNE для аналізу 10,000 клітин:

- Вхідні дані: матриця експресії 20,000 генів × 10,000 клітин
- Попередня обробка: PCA до 50 компонент.

Приклад 2: Візуалізація простору ознак CNN

Компанія розробляє систему розпізнавання дефектів:

- Витягнення ознак з передостаннього шару ResNet-50 - 2048-вимірні вектори для 50,000 зображень продукції

Приклад 3: Аналіз клієнтської поведінки

Банк аналізує транзакційні дані клієнтів: - 100 ознак поведінки для 1 мільйона клієнтів - Barnes-Hut t-SNE для масштабованості.

Переваги та недоліки

Переваги

Виявлення нелінійних структур

- Здатність знаходити складні многовиди
- Краще за лінійні методи для складних даних
- Збереження локальної структури

Візуальна якість

- Чіткі, добре розділені кластери
- Інтуїтивно зрозумілі візуалізації
- Виявлення прихованих патернів

Гнучкість

- Працює з різними типами даних
- Адаптується до локальної щільності
- Налаштування через параметр perplexity

Робастність

- Стійкість до шуму в даних
- Менша чутливість до викидів порівняно з PCA
- Збереження топології даних

Недоліки

Обчислювальна складність

- $O(n^2)$ для стандартної версії
- Повільно для великих датасетів
- Високі вимоги до пам’яті

Недетермінованість

- Різні запуски дають різні результати
- Залежність від початкової ініціалізації
- Складність відтворення результатів

Збереження глобальної структури

- Відстані між кластерами можуть бути спотворені
- Неможливість інтерпретації абсолютних відстаней
- Фокус на локальній структурі

Складність вибору параметрів

- Чутливість до міри невизначеності

- Немає універсальних правил вибору

- Необхідність експериментування

Обмеження застосування

- Тільки для візуалізації, не для подальшого аналізу

- Неможливість проекції нових точок

- Відсутність зворотного перетворення

Контрольні запитання

1. У чому полягає основна відмінність між t-SNE та PCA з точки зору збереження структури даних?

2. Поясніть, чому t-SNE використовує різні розподіли (гаусів та t-розподіл) для високовимірного та низьковимірного просторів?

3. Що таке perplexity та як цей параметр впливає на результат візуалізації?

4. Яка роль KL-дивергенції в алгоритмі t-SNE?

5. Чому t-SNE краще підходить для візуалізації, ніж для попередньої обробки даних у машинному навчанні?

6. Як Barnes-Hut апроксимація покращує масштабованість t-SNE?

7. Що таке “early exaggeration” та яка його роль в алгоритмі?

8. Чому важливо нормалізувати дані перед застосуванням t-SNE?

9. Які існують альтернативи t-SNE для нелінійного зниження розмірності?

10. Як інтерпретувати відстані між кластерами в t-SNE візуалізації?

Методичні вказівки до лабораторної роботи

1. Назва

Реалізація та дослідження алгоритму t-SNE для візуалізації багатовимірних даних

2. Мета

Вивчити теоретичні основи алгоритму t-SNE, реалізувати базову версію, дослідити вплив параметрів на результат візуалізації та порівняти з іншими методами зниження розмірності.

3. Завдання

- Реалізувати базовий алгоритм t-SNE
- Дослідити вплив параметра perplexity
- Візуалізувати процес оптимізації
- Порівняти з PCA та UMAP
- Застосувати до реальних датасетів
- Реалізувати прискорення Barnes-Hut

4. Опис

Лабораторна робота передбачає покрокову реалізацію алгоритму t-SNE, починаючи з обчислення матриць ймовірностей та закінчуючи візуалізацією результатів. Студенти досліджують вплив різних параметрів та порівнюють результати з іншими методами.

5. Порядок виконання

1. Підготовка тестових даних (синтетичні кластери)
2. Реалізація обчислення матриці ймовірностей P
3. Реалізація пошуку оптимальних σ_i
4. Реалізація обчислення матриці Q
5. Реалізація градієнтного спуску з momentum
6. Візуалізація проміжних результатів
7. Дослідження впливу perplexity (5, 15, 30, 50)
8. Застосування до датасету MNIST
9. Порівняння часу виконання з sklearn.manifold.TSNE
10. Аналіз та інтерпретація результатів

6. Результат

Робоча реалізація t-SNE з можливістю візуалізації багатовимірних даних, графіки залежності якості від параметрів, порівняльний аналіз з іншими методами.

Після виконання лабораторної роботи студент повинен надати:

1. Вихідний код програми, де чітко виділені:
- Функція обчислення матриці ймовірностей P
- Функція бінарного пошуку для σ_i
- Функція обчислення матриці Q

- Функція обчислення градієнта
- Основний цикл оптимізації
- Функції візуалізації

2. Продемонструвати роботу додатку:
- Візуалізація синтетичних даних
- Анімація процесу оптимізації
- Результати для різних значень міри невизначенності
- Візуалізація MNIST датасету
- Графіки збіжності (KL-дивергенція від ітерації)

3. Бути готовим відповісти на питання:
- Як вибрати оптимальне значення perplexity?
- Чому результати різняться між запусками?
- Як інтерпретувати отримані візуалізації?
- Які обмеження має реалізований алгоритм?

Тести

1. t-SNE оптимізує:

1) Евклідові відстані між точками
2) KL-дивергенцію між розподілами ймовірностей
3) Дисперсію даних
4) Кореляцію між ознаками

2. У низьковимірному просторі t-SNE використовує:

1) Гаусів розподіл
2) Рівномірний розподіл
3) t-розподіл Стьюдента
4) Експоненційний розподіл

3. Параметр perplexity приблизно відповідає:

1) Розмірності вихідного простору
2) Кількості ітерацій
3) Ефективній кількості сусідів
4) Швидкості навчання

4. Обчислювальна складність стандартного t-SNE:

1) O(n)

2) O(n log n)

3) $O(n^2)$

4) $O(n^3)$

5. Early exaggeration в t-SNE:

1) Зменшує кількість ітерацій

2) Допомагає формуванню чітких кластерів на початку

3) Прискорює збіжність

4) Зменшує міру невизначеності

Відповіді:

1-2,

2-3,

3-3,

4-3,

5-2

Тести підвищеної складності

1. Barnes-Hut апроксимація в t-SNE базується на:

1) Методі головних компонент

2) Використанні дерев для групування далеких точок

3) Стохастичному градієнтному спуску

4) Ядровий трюк

2. Основна причина використання t-розподілу замість гаусового:

1) Швидші обчислення

2) Кращі теоретичні властивості

3) Важчі хвости для кращого розділення кластерів

4) Простіша реалізація

3. Параметр θ в Barnes-Hut t-SNE контролює:

1) Швидкість навчання

2) Компроміс між швидкістю та точністю

3) Розмір кластерів

4) Кількість ітерацій

4. Для датасету з n точок, матриця P має властивість:

1) Симетричність та сума елементів = 1

2) Ортогональність

3) Діагональність

4) Розрідженість

5. Градієнт KL-дивергенції в t-SNE має вигляд:

1) Лінійної функції відстаней

2) Суми привабливих та відштовхувальних сил

3) Константи

4) Експоненційної функції

6. FIt-SNE досягає прискорення через:

1) GPU обчислення

2) Швидке перетворення Фур'є для згортки

3) Паралелізацію

4) Зменшення кількості параметрів

7. При збільшенні міри невизначеності:

1) Збільшується локальна структура

2) Збільшується врахування глобальної структури

3) Зменшується час обчислень

4) Покращується збіжність

8. параметричний t-SNE відрізняється можливістю:

1) Використання різних метрик

2) Проекції нових точок через навчену функцію

3) Збереження глобальних відстаней

4) Зворотного перетворення

9. Crowding problem в оригінальному SNE вирішується через:

1) Збільшення розмірності

2) Використання різних розподілів для P та Q

3) Зменшення міри невизначеності

4) Додавання регуляризації

10. Art-SNE розширення дозволяє:

1) Прискорити обчислення

2) Контролювати відстані між конкретними точками

3) Автоматично вибирати параметри

4) Працювати з категоріальними даними

Відповіді:

1. 2 - Barnes-Hut використовує квадродерева/октодерева

2. 2 - t-розподіл має важчі хвости, що запобігає стисненню

3. 2 - θ визначає точність апроксимації (зазвичай 0.5)

4. 1 - P симетрична та нормалізована

5. 2 - Градієнт як баланс притягання та відштовхування

6. 2 - FFT для обчислення ядрових сум

7. 2 - Більша міра невизначеності враховує більше сусідів

8. 2 - Параметрична версія навчає функцію відображення

9. 2 - Різні розподіли дають більше простору в низьких вимірах

10. 2 - Art-SNE дозволяє інтерактивно керувати візуалізацією

Завдання для самостійної роботи

1. Реалізуйте модифікацію t-SNE з адаптивною мірою невизначеності для різних регіонів даних.

2. Дослідіть вплив різних функцій відстані (манхеттенська, косинусна) на результати t-SNE.

3. Створіть інтерактивну візуалізацію, що показує еволюцію t-SNE в реальному часі.

4. Реалізуйте multi-scale t-SNE, що створює ієрархію візуалізацій.

5. Порівняйте різні стратегії ініціалізації (випадкова, PCA, Laplacian Eigenmaps).

6. Дослідіть можливість використання t-SNE для виявлення викидів у даних.

7. Реалізуйте dynamic t-SNE для візуалізації даних, що змінюються в часі.

8. Створіть ансамбль t-SNE візуалізацій для оцінки стабільності кластерів.

9. Дослідіть використання різних оптимізаторів (Adam, RMSprop) замість momentum SGD.

10. Розробіть метрики для кількісної оцінки якості t-SNE візуалізації.

Теми лабораторних робіт

1. Базова реалізація t-SNE та візуалізація класичних ML датасетів.

2. Порівняльний аналіз t-SNE, PCA та UMAP для різних типів даних.

3. Використання t-SNE для аналізу представлень глибоких нейронних мереж.

4. Застосування Barnes-Hut t-SNE до великих датасетів (>100K точок).

5. Дослідження parametric t-SNE з використанням нейронних мереж.

6. Візуалізація еволюції навчання нейронної мережі через t-SNE.

7. Комбінування t-SNE з кластеризацією для автоматичного виявлення груп.

8. Розробка веб-інтерфейсу для інтерактивної t-SNE візуалізації.

9. Застосування t-SNE до аналізу текстових систем (Word2Vec, BERT).

10. дослідження розподілів із важкими хвостами (heavy-tailed) t-SNE для покращення розділення кластерів.

Розділ 16: Теорія обчислюваності та складності

Теорія обчислюваності та складності становить фундаментальну основу інформатики, відповідаючи на найглибші питання про природу обчислень, межі можливого та ефективність алгоритмів. Цей розділ присвячений вивченню теоретичних аспектів, які визначають, що можна обчислити взагалі, а що можна обчислити ефективно.

Розпочинаємо з машин Тюрінга - універсальної моделі обчислень, запропонованої Аланом Тюрінгом у 1936 році. Ця елегантна абстракція не лише формалізує поняття алгоритму, але й дозволяє довести існування задач, які принципово неможливо розв’язати жодним алгоритмом. Вивчення нерозв’язних задач розкриває фундаментальні обмеження обчислень та має глибокі філософські наслідки для розуміння природи математики та логіки.

Далі переходимо до теорії складності обчислень, яка класифікує задачі за ресурсами, необхідними для їх розв'язання. Центральними є класи P та NP, що відповідають задачам з поліноміальним часом розв'язання та перевірки відповідно. Питання рівності P та NP залишається найвідомішою відкритою проблемою в інформатиці, а концепція NP-повноти дозволяє ідентифікувати найскладніші задачі в класі NP. Клас PSPACE розширює нашу перспективу, охоплюючи задачі, що потребують поліноміальної пам'яті.

Параметризована складність пропонує більш тонкий погляд на обчислювальну складність, виділяючи параметри задачі та досліджуючи алгоритми, ефективні при малих значеннях цих параметрів. Клас FPT (fixed-parameter tractable) містить задачі, які можна ефективно розв'язати для малих параметрів, навіть якщо загальна задача є NP-складною.

Завершуємо розділ вивченням online-алгоритмів, які приймають рішення без повної інформації про майбутні вхідні дані. Конкурентний аналіз дозволяє оцінити якість таких алгоритмів порівняно з оптимальним offline-алгоритмом, що має повну інформацію. Ця тема особливо актуальна для сучасних розподілених систем та алгоритмів реального часу.

Матеріал цього розділу формує теоретичний фундамент для розуміння можливостей та обмежень обчислень, надаючи інструменти для аналізу складності алгоритмів та проектування ефективних рішень складних задач.

Машини Тюрінга та нерозв'язні задачі.

Вступ до теорії обчислюваності

Теорія обчислюваності є фундаментальною галуззю теоретичної інформатики та математичної логіки, яка вивчає принципові можливості та обмеження обчислень. Центральне питання цієї теорії полягає в тому, які задачі можна розв'язати алгоритмічно, а які принципово не піддаються алгоритмічному розв'язанню.

Історія теорії обчислюваності бере початок у 1930-х роках, коли математики намагалися формалізувати поняття алгоритму та обчислення. Давид Гільберт у своїй знаменитій програмі поставив питання про можливість створення універсального методу розв'язання математичних проблем. Відповідь на це питання привела до революційних відкриттів у математиці та заклала основи сучасної інформатики.

Алан Тюрінг у 1936 році запропонував абстрактну обчислювальну модель, яка згодом отримала назву машини Тюрінга. Ця модель стала основою для формального визначення поняття алгоритму та обчислюваності. Приблизно в той же час Алонзо Черч розробив λ-числення, а Еміль Пост створив свою модель обчислень. Усі ці підходи виявилися еквівалентними, що привело до формулювання тези Черча-Тюрінга.

Теза Черча-Тюрінга стверджує, що будь-яка функція, яку можна обчислити за допомогою алгоритму, може бути обчислена машиною Тюрінга. Хоча ця теза не може бути доведена математично (оскільки поняття “алгоритм” у ній використовується в інтуїтивному сенсі), вона підтверджується тим фактом, що всі відомі формалізації поняття алгоритму виявилися еквівалентними.

Обчислюваність і моделі обчислень

У математиці здавна існувало інтуїтивне уявлення про алгоритм – точний набір інструкцій (правил), що визначають послідовність операцій для розв’язання задач певного типу. Протягом століть накопичено безліч алгоритмів для різних обчислювальних задач:

- Алгоритм Евкліда (бл. 300 р. до н.е.) – знаходження найбільшого спільного дільника
- Решето Ератосфена (бл. 200 р. до н.е.) – знаходження простих чисел
- Метод Гауса (1810) – розв’язання систем лінійних рівнянь
- Алгоритм Евклідового ділення – ділення “кутом”

Математики довгий час вважали, що будь-яку математичну проблему в принципі можна розв’язати алгоритмічно (механічно) – треба лише віднайти відповідний метод.

Програма Гільберта

Цю віру яскраво висловив Давид Гільберт (1862-1943) у своїй знаменитій програмі формалізації математики. На Міжнародному конгресі математиків 1900 року він сформулював 23 проблеми, серед яких:

- Друга проблема: доведення несуперечності арифметики
- Десята проблема: існування загального методу (алгоритму) розв’язання будь-якого діофантового рівняння

У 1928 році Гільберт сформулював Entscheidungsproblem (проблему розв'язності) – питання про існування алгоритму, який би визначав істинність довільного твердження в логіці першого порядку.

Лише у XX столітті виявилося, що такий універсальний метод може взагалі не існувати:

1931 – Курт Гедель довів теореми про неповноту, показавши існування істинних але недоведених тверджень

1936 – Алан Тюрінг та Алонзо Черч незалежно довели нерозв'язність Entscheidungsproblem

1970 – Юрій Матіясевич довів нерозв'язність десятої проблеми Гільберта

Для строгої відповіді на питання про межі алгоритмічної розв'язності знадобилося формалізувати поняття алгоритму. До 1930-х років не існувало точного математичного означення алгоритму – його розуміли інтуїтивно.

Починаючи з середини 1930-х було запропоновано кілька еквівалентних формалізацій:

12. λ-числення Алонзо Черча (1936)
13. Машини Тюрінга Алана Тюрінга (1936)
14. Рекурсивні функції Курта Геделя та Стівена Кліні
15. Нормальні алгоритми Андрія Маркова (1940-ті)
16. Машини Поста Еміля Поста (1936)

Машина Тюрінга: формальне означення і принцип роботи

Машина Тюрінга – це абстрактний обчислювальний пристрій, що моделює роботу ідеалізованого комп'ютера з необмеженою пам'яттю. Вона складається з:

17. Стрічки – потенційно нескінченної в обидва боки
18. Головки читання/запису
19. Керуючого пристрою зі скінченною кількістю станів
20. Програми (таблиці переходів)

Означення 2.1. Детермінована машина Тюрінга – це сімка:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$$

де:

– Q – скінченна множина станів

- Σ – вхідний алфавіт (не містить символу пробілу)
- Γ – стрічковий алфавіт, де $\Sigma \subset \Gamma$ і $\blacksquare \in \Gamma$ (символ пробілу)
- $\delta: Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$ – функція переходів
- $q_0 \in Q$ – початковий стан
- $q_{accept} \in Q$ – приймаючий стан
- $q_{reject} \in Q$ – відхиляючий стан, де $q_{accept} \neq q_{reject}$

Означення 2.2. Конфігурація машини Тюрінга – це трійка (q, w, i), де: - $q \in Q$ – поточний стан - $w \in \Gamma^*$ – вміст стрічки - $i \in \mathbb{N}$ – позиція головки

Початкова конфігурація для входу $x \in \Sigma^*$: $(q_0, x, 0)$

Означення 2.3. Крок обчислення – це перехід між конфігураціями згідно з функцією δ:

Якщо $\delta(q, a) = (q', b, D)$, то: - $(q, uav, i) \vdash (q', ubv, i')$, де: - $i' = i - 1$ якщо $D = L$ - $i' = i + 1$ якщо $D = R$ - $i' = i$ якщо $D = S$

Означення 2.4. Обчислення машини M на вході x – це послідовність конфігурацій:

$$C_0 \vdash C_1 \vdash C_2 \vdash \ldots \vdash C_n$$

де C_0 – початкова конфігурація, C_n – кінцева (якщо існує).

Означення 2.5. Машина M приймає слово x, якщо існує обчислення, що завершується в стані q_{accept}.

Означення 2.6. Мова, що розпізнається машиною M:

$$L(M) = \{x \in \Sigma^* : M \text{ приймає } x\}$$

Означення 2.7. Мова L є:

- Рекурсивно перелічуваною (р.п.), якщо існує машина Тюрінга M така, що $L = L(M)$

- Рекурсивною (розв'язною), якщо існує машина Тюрінга M, яка зупиняється на всіх входах і $L = L(M)$

Приклад 1: Розпізнавання мови $\{0^n 1^n : n \geq 0\}$

Ідея алгоритму:

1. Позначити крайній лівий 0 символом X

2. Рухатися вправо до першої 1, позначити її Y

3. Повернутися до наступного 0 і повторити

4. Якщо всі 0 та 1 позначені – прийняти

Формальний опис:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_{accept}, q_{reject}\}$

- $\Sigma = \{0,1\}$ - $\Gamma = \{0,1, X, Y, \blacksquare\}$

Таблиця переходів:

Стан	Символ	Новий стан	Новий символ	Напрям
q_0	0	q_1	X	R
q_0	Y	q_3	Y	R
q_0	▪	q_{accept}	▪	S
q_1	0	q_1	0	R
q_1	1	q_2	Y	L
q_1	Y	q_1	Y	R
q_2	0	q_2	0	L
q_2	X	q_0	X	R
q_2	Y	q_2	Y	L
q_3	Y	q_3	Y	R
q_3	▪	q_{accept}	▪	S

Приклад роботи на вході “0011”:

$q_0 0011 \vdash Xq_1 011 \vdash X0q_1 11 \vdash Xq_2 0Y1 \vdash q_2 X0Y1 \vdash Xq_0 0Y1 \vdash XXq_1 Y1 \vdash XXYq_1 1 \vdash XXq_2 YY \vdash Xq_2 XYY \vdash XXq_0 YY \vdash XXYq_3 Y \vdash XXYYq_3 □ \vdash XXYY□q\_\{accept\}$

Приклад 2: Додавання 1 до двійкового числа

Ідея алгоритму:

1. Перейти в кінець числа

2. Рухатися вліво, замінюючи 1 на 0 (перенос)

3. Першу 0 замінити на 1

4. Якщо дійшли до початку – дописати 1

Формальний опис:

- $Q = \{q_0, q_1, q_2, q_{accept}\}$

- $\Sigma = \{0,1\}$ - $\Gamma = \{0,1, \blacksquare\}$

Таблиця переходів:

Стан	Символ	Новий стан	Новий символ	Напрям
q_0	0	q_0	0	R
q_0	1	q_0	1	R
q_0	▫	q_1	▫	L
q_1	0	q_{accept}	1	S
q_1	1	q_1	0	L
q_1	▫	q_2	▫	R
q_2	0	q_{accept}	1	S

Приклад роботи на вході “1011”:

$q_0 1011 \vdash 1q_0 011 \vdash 10q_0 11 \vdash 101q_0 1 \vdash 1011q_0 \square \vdash 101q_1 1 \vdash 10q_1 10 \vdash 1q_1 0$
$00 \vdash q_1 1000 \vdash q_1 \square 0000 \vdash \square q_2 0000 \vdash \square 1q_{accept} 000$

Результат: 10000 (додали 1 до 1011)

Приклад 3: Подвоєння унарного числа

Завдання: Перетворити 1^n на 1^{2n}

Ідея алгоритму:

1. Замінювати кожну 1 на X

2. Дописувати дві 1 в кінець

3. Повернути X назад на 1

Формальний опис:

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_{accept}\}$

- $\Sigma = \{1\}$ - $\Gamma = \{1, X, Y, ▫\}$

Таблиця переходів:

Стан	Символ	Новий стан	Новий символ	Напрям
q_0	1	q_1	X	R
q_0	Y	q_4	Y	R
q_0	▫	q_{accept}	▫	S
q_1	1	q_1	1	R
q_1	Y	q_1	Y	R
q_1	▫	q_2	Y	R
q_2	▫	q_3	1	L

Стан	Символ	Новий стан	Новий символ	Напрям
q_3	Y	q_0	Y	L
q_4	Y	q_4	1	R
q_4	▫	q_{accept}	▫	S

Варіанти машин Тюрінга

1. Багатострічкові машини Тюрінга

Означення 1. k-стрічкова машина Тюрінга має:

- k стрічок, кожна зі своєю головкою
- Функція переходів: $\delta: Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$

Теорема 1. Для кожної k-стрічкової машини існує еквівалентна одностічкова машина.

2. Недетерміновані машини Тюрінга

Означення 2. Недетермінована машина Тюрінга має функцію переходів:

$$\delta: Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R,S\}}$$

Теорема 2. Класи мов, що розпізнаються детермінованими та недетермінованими машинами Тюрінга, збігаються.

3. Машини Тюрінга з напівнескінченною стрічкою

Означення 3. Машина з напівнескінченною стрічкою має стрічку, обмежену зліва.

4. Лінійно обмежені автомати

Означення 4. Лінійно обмежений автомат – це машина Тюрінга, яка використовує лише $O(n)$ комірок стрічки для входу довжини n.

Універсальна машина Тюрінга

1. Кодування машин Тюрінга

Для побудови універсальної машини потрібно закодувати довільну машину Тюрінга як рядок.

Схема кодування:

1. Пронумерувати стани: $q_1, q_2, \ldots, q_n$
2. Пронумерувати символи: $a_1, a_2, \ldots, a_m$
3. Кодувати напрямки: $L = 1, R = 2, S = 3$

4. Кожне правило $\delta(q_i, a_j) = (q_k, a_l, d)$ кодується як:

$$0^i 10^j 10^k 10^l 10^d$$

5. Правила розділяються “11”

Приклад: Правило $\delta(q_2, 0) = (q_3, 1, R)$ кодується як 0010010001010011

Означення 1. Універсальна машина Тюрінга U – це машина, яка на вході $\langle M, x\rangle$ симулює роботу машини M на вході x:

$$U(\langle M, x\rangle) = \begin{cases} M(x), & \text{якщо } M \text{ зупиняється на } x \\ \text{не зупиняється}, & \text{інакше} \end{cases}$$

Теорема про існування універсальної машини

Теорема (Тюрінг, 1936). Існує універсальна машина Тюрінга.

Універсальна машина працює за таким алгоритмом:

1. Розділити вхід на код машини та дані

2. Ініціалізувати симуляцію

3. Циклічно виконувати:

- Знайти правило для поточного стану та символу
- Виконати відповідну дію
- Оновити конфігурацію

4. Якщо досягнуто кінцевого стану – завершити

Історичне значення

Концепція універсальної машини Тюрінга:

- Передбачила архітектуру фон Неймана (1945)
- Стала основою для створення програмованих комп’ютерів
- Показала можливість зберігання програм як даних

Проблема зупинки та нерозв’язні задачі

1. Проблема зупинки

Означення 1. Проблема зупинки:

$$HALT = \{\langle M, x\rangle : \text{машина } M \text{ зупиняється на вході } x\}$$

Теорема (Тюрінг, 1936). Проблема зупинки є нерозв’язною.

Ідея доведення:

1. Припустимо, існує машина H, яка розв’язує HALT

2. Побудуємо машину D:

```
D(⟨M⟩):
if H(⟨M, ⟨M⟩⟩) = "зупиняється":
loop forever      else:
halt
```

3. Розглянемо $D(\langle D \rangle)$:

- Якщо D зупиняється на $\langle D \rangle$, то H каже “зупиняється”, отже D зациклюється

- Якщо D не зупиняється на $\langle D \rangle$, то H каже “не зупиняється”, отже D зупиняється 4.

Протиріччя!

2. Метод діагоналізації

Доведення використовує метод діагоналізації Кантора, адаптований для машин Тюрінга.

Таблиця поведінки машин:

	M_1	M_2	M_3	…	D	…
$\langle M_1 \rangle$	✓	✗	✓	…	?	…
$\langle M_2 \rangle$	✗	✓	✗	…	?	…
$\langle M_3 \rangle$	✓	✓	✗	…	?	…
…	…	…	…	…	…	…
$\langle D \rangle$	?	?	?	…	✗✓	…

3. Інші нерозв’язні проблеми

Теорема (Райс, 1953). Будь-яка нетривіальна властивість рекурсивно перелічуваних мов є нерозв’язною.

Означення 2. Властивість P мов є нетривіальною, якщо:

- Існує машина M_1 така, що $L(M_1)$ має властивість P

- Існує машина M_2 така, що $L(M_2)$ не має властивості P

Приклади нерозв’язних проблем:

Проблема порожньої мови:

$$EMPTY = \{\langle M \rangle : L(M) = \emptyset\}$$

Проблема еквівалентності:

$$EQ = \{\langle M_1, M_2\rangle : L(M_1) = L(M_2)\}$$

Проблема регулярності:

$$REG = \{\langle M\rangle : L(M) \text{ є регулярною}\}$$

Проблема Поста про відповідність (PCP): Дано набір доміно $\{(u_1, v_1), \ldots, (u_n, v_n)\}$. Чи існує послідовність індексів така, що:

$$u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}$$

Десята проблема Гільберта: Чи має діофантове рівняння розв'язки в цілих числах?

Теза Черча

Теза Черча: Клас функцій, обчислюваних машинами Тюрінга, збігається з класом інтуїтивно обчислюваних функцій.

Всі наступні моделі мають однакову обчислювальну потужність:

1. Машини Тюрінга
2. λ-числення
3. Рекурсивні функції
4. Машини Поста
5. Нормальні алгоритми Маркова
6. RAM-машини
7. Сучасні мови програмування

Розширена теза Черча-Тюрінга

Теза Черча-Тюрінга: Будь-який фізично реалізований процес обчислення може бути змодельований машиною Тюрінга.

Відкрите питання: чи можуть квантові комп'ютери обчислювати щось, недоступне класичним машинам Тюрінга?

Задачі для самостійного розв'язання

1. Побудуйте машину Тюрінга, яка подвоює унарне число (замінює $\mathtt{1}^n$ на $\mathtt{1}^{2n}$).

2. Доведіть, що мова $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$ розпізнається машиною Тюрінга.

3. Покажіть, що двострічкова машина Тюрінга може симулювати одно стрічкову з квадратичним уповільненням.

4. Доведіть, що проблема "Чи приймає машина Тюрінга хоча б один вхід?" є нерозв'язною.

5. Використовуючи теорему Райса, доведіть нерозв'язність проблеми "Чи є мова, що розпізнається машиною Тюрінга, скінченною?"

6. Покажіть, що множина {n | n-та машина Тюрінга зупиняється на порожньому вході} належить до Σ_1, але не до Π_1.

7. Доведіть, що якщо P ≠ NP, то існують проблеми в NP, які не є ні в P, ні NP-повними.

8. Побудуйте недетерміновану машину Тюрінга для розпізнавання мови {ww | w ∈ {0,1}*} за лінійний час.

9. Доведіть, що проблема еквівалентності для скінченних автоматів є розв'язною, на відміну від машин Тюрінга.

10. Покажіть, що існує незліченна кількість мов, але лише зліченна кількість машин Тюрінга. Що це означає для обчислюваності?

Класи складності

Класи складності: P, NP, NP-complete, PSPACE

Від обчислюваності до ефективності

Теорія обчислюваності, яку ми розглядали раніше, відповідає на фундаментальне питання: що можна обчислити в принципі? Машини Тюрінга показали нам межі можливого – існують задачі, які жоден алгоритм не може розв'язати.

Але знати, що задача розв'язна в принципі, часто недостатньо. Уявіть алгоритм, який працює 2^{1000} років – формально він існує, але практично марний. Тому виникає наступне питання: як швидко можна розв'язати задачу?

Теорія складності вивчає ресурси (час, пам'ять), необхідні для розв'язання обчислювальних задач. Вона класифікує задачі не за принципом "розв'язна чи ні", а за принципом "наскільки важко розв'язати".

Розглянемо два алгоритми сортування:

- Бульбашкове сортування: час $O(n^2)$

- Швидке сортування: час $O(n\log n)$

Для $n = 1000000$ елементів:

- Бульбашкове: $\approx 10^{12}$ операцій

- Швидке: $\approx 2 \times 10^7$ операцій

Різниця – у 50000 разів! На сучасному комп'ютері це різниця між секундами та годинами роботи.

Основні ресурси

Часова складність – кількість елементарних кроків алгоритму:

$$TIME(f(n)) = \{L: \exists \text{ МТ } M \text{ така, що } L(M) = L \text{ і } T_M(n) \leq f(n)\}$$

Просторова складність – кількість використаних комірок пам'яті:

$$SPACE(f(n)) = \{L: \exists \text{ МТ } M \text{ така, що } L(M) = L \text{ і } S_M(n) \leq f(n)\}$$

де $T_M(n)$ та $S_M(n)$ – час та простір роботи машини M на входах довжини n.

Клас P: Ефективно розв'язувані задачі

Клас P об'єднує задачі, які можна розв'язати "швидко" – за поліноміальний час. Чому саме поліном?

Теза Кобхема-Едмондса: Задачі з поліноміальними алгоритмами є практично розв'язуваними, а з експоненційними – ні.

Порівняємо характер зростання функцій для $n = 100$:

- $n = 100$

- $n^2 = 10{,}000$

- $n^3 = 1{,}000{,}000$

- $2^n \approx 10^{30}$ (більше, ніж атомів на Землі!)

Означення 2.1. Клас P (Polynomial time) – це множина мов, для яких існує детермінована машина Тюрінга та поліном $p(n)$ такі, що:

$$P = \bigcup_{k=1}^{\infty} TIME(n^k)$$

Еквівалентно:

$$L \in P \Leftrightarrow \exists \text{ МТ } M, \exists k \in \mathbb{N}: \forall x \in \Sigma^*, T_M(x) \leq |x|^k + k$$

Приклади задач у класі P

1. Пошук у графі (REACHABILITY)

Задача: Дано граф $G = (V, E)$ та вершини $s, t \in V$. Чи існує шлях від s до t?

Алгоритм (пошук у ширину, BFS):

```
BFS(G, s, t):
    Q = [s]  // черга
    visited = {s}
    while Q не порожня:
```

```
        v = Q.dequeue()
        if v == t:
            return TRUE
        for u in neighbors(v):
            if u not in visited:
                visited.add(u)
                Q.enqueue(u)
    return FALSE
```

Складність: $O(|V| + |E|)$ – лінійна від розміру графа.

2. Найбільший спільний дільник (GCD)

Задача: Дано два числа $a, b \in \mathbb{N}$. Знайти $\gcd(a, b)$.

Алгоритм Евкліда:

```
GCD(a, b):
    while b ≠ 0:
        r = a mod b
        a = b
        b = r
    return a
```

Складність: $O(\log\min(a, b))$ – логарифмічна від розміру чисел.

3. Лінійне програмування (LP)

Задача:

$$\text{Максимізувати: } c^T x$$
$$\text{За умов: } Ax \leq b$$
$$x \geq 0$$

Алгоритми:

- Симплекс-метод (експоненційний у гіршому випадку, але швидкий на практиці)

- Метод еліпсоїдів (Хачіян, 1979) – $O(n^4 L)$, де L – розмір вхідних даних

- Метод внутрішньої точки (Кармаркар, 1984) – $O(n^{3.5} L)$

4. Перевірка простоти (PRIMES)

Задача: Дано число n. Чи є n простим?

Довгий час не було відомо, чи PRIMES $\in P$. У 2002 році Агравал, Каял та Саксена довели:

Теорема (AKS, 2002): PRIMES $\in P$. Існує алгоритм з часом $O(\log^6 n)$.

Властивості класу P

Теорема Клас P замкнутий відносно:

1. Об’єднання: $L_1, L_2 \in P \Rightarrow L_1 \cup L_2 \in P$

2. Перетину: $L_1, L_2 \in P \Rightarrow L_1 \cap L_2 \in P$

3. Доповнення: $L \in P \Rightarrow \overline{L} \in P$

4. Конкатенації: $L_1, L_2 \in P \Rightarrow L_1 \cdot L_2 \in P$

5. Зірки Кліні: $L \in P \Rightarrow L^* \in P$

Доведення (для об’єднання): Нехай M_1 розпізнає L_1 за час n^{k_1}, M_2 розпізнає L_2 за час n^{k_2}. Побудуємо M:

```
M(x):
    Запустити M₁(x)
    if M₁ приймає:
        return ACCEPT
    Запустити M₂(x)
    if M₂ приймає:
        return ACCEPT
    return REJECT
```

Час роботи: $O(n^{k_1} + n^{k_2}) = O(n^{\max(k_1,k_2)})$ – поліноміальний.

Клас NP: Задачі з ефективною перевіркою

Розглянемо дві задачі:

1. Знайти розклад числа 1591 на прості множники

2. Перевірити, що 1591 = 37 × 43

Перша задача може бути складною, друга – тривіальна (одне множення).

Клас NP формалізує цю ідею: задачі, де розв’язок важко знайти, але легко перевірити.

Недетермінована машина Тюрінга

Означення 1. Недетермінована машина Тюрінга (НМТ) – це машина, яка на кожному кроці може “вгадувати” один з кількох можливих переходів.

Формально: $\delta: Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}$

НМТ приймає вхід, якщо існує хоча б одна послідовність виборів, що веде до прийняття.

Означення 2. Клас NP (Nondeterministic Polynomial time):

$$NP = \bigcup_{k=1}^{\infty} NTIME(n^k)$$

де $NTIME(f(n))$ – мови, розпізнавані НМТ за час $O(f(n))$.

Еквівалентне означення через верифікатори:

$L \in NP$ тоді і тільки тоді, коли існує поліноміальний верифікатор V та поліном $p(n)$ такі, що:

$$x \in L \Leftrightarrow \exists y \in \{0,1\}^{p(|x|)}: V(x,y) = 1$$

де: - x – вхід - y – сертифікат (свідок, доказ) - V працює за поліноміальний час

Приклади задач у NP

1. SAT (Булева виконуваність)

Вхід: Булева формула ϕ в КНФ (кон'юнктивній нормальній формі):

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge \ldots$$

Питання: Чи існує набір значень змінних, при якому $\phi = \text{TRUE}$?

Сертифікат: Конкретні значення змінних.

Верифікація: Підставити значення та обчислити – $O(|\phi|)$.

2. Гамільтонів цикл (HAMCYCLE)

Вхід: Граф $G = (V, E)$.

Питання: Чи існує цикл, що проходить через кожну вершину рівно один раз?

Сертифікат: Послідовність вершин $(v_1, v_2, \ldots, v_n, v_1)$.

Верифікація:

```
Перевірити, що всі вершини різні
Перевірити, що (vᵢ, vᵢ₊₁) ∈ E для всіх i
Перевірити, що (vₙ, v₁) ∈ E
```

Час: $O(n)$.

3. Розклад на множники (FACTORING)

Вхід: Числа $n, k \in \mathbb{N}$.

Питання: Чи має n дільник d такий, що $1 < d < k$?

Сертифікат: Число d.

Верифікація: Перевірити, що $d|n$ та $1 < d < k - O(\log n)$.

4. Задача про рюкзак (KNAPSACK)

Вхід: - Предмети з вагами $w_1, \ldots, w_n$ та цінностями $v_1, \ldots, v_n$ - Місткість рюкзака W - Цільова цінність V

Питання: Чи можна вибрати предмети з сумарною вагою $\leq W$ та цінністю $\geq V$?

Сертифікат: Підмножина предметів $S \subseteq \{1, \ldots, n\}$.

Верифікація:

$$\sum_{i \in S} w_i \leq W \text{ та } \sum_{i \in S} v_i \geq V$$

Співвідношення P та NP

Теорема 1. $P \subseteq NP$.

Проблема тисячоліття: Чи $P = NP$?

Це одна з семи проблем тисячоліття з премією $1,000,000.

Аргументи за $P \neq NP$:

- Десятиліття безуспішних спроб
- Інтуїтивна різниця між пошуком та перевіркою
- Наслідки для криптографії були б катастрофічними

Аргументи за $P = NP$:

- Немає доведення протилежного
- Можливо, ми просто не знайшли правильного підходу

Поліноміальні зведення

Ідея зведення

Зведення – це спосіб порівняти складність задач. Якщо задачу A можна звести до задачі B, то B не простіша за A.

Аналогія: Якщо ви вмієте готувати торт (задача B), і я можу перетворити приготування печива (задача A) на приготування торта, то приготування торта не простіше за приготування печива.

Означення 4.1. Поліноміальне зведення (зведення Карпа):

Мова A поліноміально зводиться до мови B (позначається $A \leq_p B$), якщо існує функція $f: \Sigma^* \to \Sigma^*$ така, що:

1. f обчислюється за поліноміальний час

2. $\forall x \in \Sigma^*: x \in A \Leftrightarrow f(x) \in B$

Властивості зведень

Теорема 1. Зведення транзитивне:

$$A \leq_p B \wedge B \leq_p C \Rightarrow A \leq_p C$$

Теорема 2. Зведення зберігає належність до P:

$$A \leq_p B \wedge B \in P \Rightarrow A \in P$$

Доведення: Нехай f – зведення від A до B, M_B розв'язує B за час n^k.

```
Алгоритм для A(x):
    y = f(x)          // час O(n^m)
    return M_B(y)     // час O(|y|^k) = O(n^mk)
```

Загальний час: $O(n^{mk})$ – поліноміальний.

Клас NP-complete: Найважчі задачі в NP

NP-повні задачі – це "найважчі" задачі в NP. Якщо ми знайдемо поліноміальний алгоритм для будь-якої з них, то $P = NP$.

Означення 1. Мова L є NP-важкою (NP-hard), якщо:

$$\forall A \in NP: A \leq_p L$$

Означення 2. Мова L є NP-повною (NP-complete), якщо: 1. $L \in NP$ 2. L є NP-важкою

Теорема Кука-Левіна

Теорема (Кук, 1971; Левін, 1973). SAT є NP-повною.

Ідея доведення:

1. SAT ∈ NP (очевидно)

2. Для довільної $L \in NP$ покажемо $L \leq_p$ SAT:

- Нехай V – верифікатор для L, працює за час n^k

- Для входу x будуємо формулу ϕ_x таку, що:

- ϕ_x виконувана ⇔ існує сертифікат y такий, що $V(x, y) = 1$

3. Конструкція ϕ_x:

- Змінні: стан машини, вміст стрічки, позиція головки в кожен момент часу

- Клози: початкова конфігурація, правила переходів, фінальне прийняття

Метод доведення NP-повноти

Після теореми Кука-Левіна, для доведення NP-повноти нової задачі достатньо:

1. Показати, що задача в NP

2. Звести відому NP-повну задачу до нової

Каталог NP-повних задач

1. Задача комівояжера (TSP)

Вхід: Повний граф з вагами ребер, число k.

Питання: Чи існує гамільтонів цикл вагою $\leq k$?

2. Розфарбування графа (COLORING)

Вхід: Граф G, число k.

Питання: Чи можна розфарбувати вершини в k кольорів так, щоб сусідні вершини мали різні кольори?

Цікавий факт: 2-COLORING $\in P$, але 3-COLORING є NP-повною!

Клас co-NP

Означення 6.1.

$$\text{co-NP} = \{L : \overline{L} \in NP\}$$

Інтуїтивно: задачі, де легко перевірити "НІ"-відповідь.

Приклади

1. TAUTOLOGY: Чи є булева формула тавтологією (істинна при всіх наборах)?

- Це co-NP задача

- Сертифікат невиконуваності: набір змінних, при якому формула хибна

2. PRIMES (до 2002): Чи є число простим?

- Сертифікат непростоти: розклад на множники

– Тепер відомо, що PRIMES $\in P$

Співвідношення NP та co-NP

Відкрите питання: Чи NP = co-NP?

Більшість вважає, що NP ≠ co-NP, бо:

- Якщо NP = co-NP, то жодна NP-повна задача не в co-NP

- Але ми не знаємо доведення

Теорема $P = NP \cap$ co-NP (якщо P = NP).

Клас PSPACE

Означення 7.1.

$$PSPACE = \bigcup_{k=1}^{\infty} SPACE(n^k)$$

Задачі, розв'язувані з поліноміальною пам'яттю.

Приклади задач у PSPACE

1. Квантифікована булева формула (QBF)

Вхід: Формула виду

$$\exists x_1 \forall x_2 \exists x_3 \dots Q_n x_n : \phi(x_1, \dots, x_n)$$

Питання: Чи істинна ця формула?

Алгоритм: Рекурсивно перебирати значення змінних.

```
QBF(φ, i):
    if i > n:
        return обчислити φ
    if Q_i = ∃:
        return QBF(φ[x_i=0], i+1) OR QBF(φ[x_i=1], i+1)
    else:  // Q_i = ∀
        return QBF(φ[x_i=0], i+1) AND QBF(φ[x_i=1], i+1)
```

Простір: $O(n)$ для стеку рекурсії.

2. Ігри на графах

Узагальнені шахи/шашки/го: На дошці $n \times n$, чи має перший гравець виграшну стратегію?

Ці задачі PSPACE-повні для достатньо великих n.

Властивості PSPACE

Теорема (Савіч, 1970). $PSPACE = NPSPACE$

Недетермінізм не дає переваги для простору!

Ієрархія включень

$$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

Що ми знаємо:

- $P \neq EXPTIME$ (теорема про часову ієрархію)
- Отже, хоча б одне включення строге
- Але яке саме – невідомо!

PSPACE-повні задачі

Означення Задача L є PSPACE-повною, якщо:

1. $L \in PSPACE$

2. $\forall A \in PSPACE : A \leq_p L$

Приклади:

- QBF (узагальнення SAT)

- Узагальнені настільні ігри

- Задачі планування в AI

Теорія складності – це не просто абстрактна математика. Вона дає нам карту обчислювального всесвіту, показуючи, де шукати ефективні алгоритми, а де – задовольнятися компромісами. Розуміння цих концепцій є критичним для кожного, хто працює з алгоритмами та обчисленнями.

Параметризована складність та FPT.

Параметризована складність та FPT

Проблема з класичною складністю

Уявімо, що ви розробляєте алгоритм для соціальної мережі. Потрібно знайти групу з k людей, які всі знають один одного (кліка). Це NP-повна задача, і найкращий відомий алгоритм працює за час $O(2^n)$, де n - кількість користувачів.

Але зачекайте! У реальних соціальних мережах: - n може бути мільйони (користувачі Facebook) - k зазвичай маленьке (групи друзів рідко перевищують 10-20 осіб)

Час $O(2^n)$ = катастрофа. Але що, якщо час буде $O(2^k \cdot n^2)$?

При $k = 10$ та $n = 1{,}000{,}000$:

- $2^{10} \cdot 10^{12} = 1024 \cdot 10^{12} \approx 10^{15}$ операцій

- На сучасному комп'ютері це години, а не століття!

Ідея параметризації

Параметризована складність розділяє вхідні дані на дві частини:

- Основний розмір n (може бути великим)

- Параметр k (часто малий на практиці)

Замість питання "Чи можна розв'язати за поліноміальний час?" ми питаємо: "Чи можна розв'язати за час $f(k) \cdot n^c$, де f - довільна функція лише від k?"

Приклади природних параметрів

1. Розмір розв'язку: знайти вершинне покриття розміру $\leq k$

2. Структурні параметри: ширина дерева графа

3. Відстань від тривіального випадку: кількість видалень до планарного графа

4. Якість апроксимації: відхилення від оптимального розв'язку

Параметризована задача

Означення 1. Параметризована задача - це мова $L \subseteq \Sigma^* \times \mathbb{N}$, де другий компонент є параметром.

Позначення: $(x, k) \in L$ означає, що екземпляр x з параметром k належить до L.

Приклад: k-VERTEX COVER

- Вхід: Граф $G = (V, E)$, число k

- Параметр: k

- Питання: Чи існує $S \subseteq V$, $|S| \leq k$, що покриває всі ребра?

Параметрична обчислювальна ефективність. (FPT)

Означення 2. Параметризована задача L належить до класу FPT (fixed-parameter tractable), якщо існують:

- Алгоритм A, що розв'язує L

- Обчислювана функція $f: \mathbb{N} \to \mathbb{N}$

- Константа c

такі, що для всіх $(x, k) \in \Sigma^* \times \mathbb{N}$:

$$\text{Час роботи } A(x, k) \leq f(k) \cdot |x|^c$$

Ключова ідея: Експоненційна складність дозволена, але лише від параметра k, не від n!

Техніки розробки FPT алгоритмів

1. Обмежене дерево пошуку (Bounded Search Tree)

Приклад: k-VERTEX COVER

Ідея: Для кожного ребра (u, v) принаймні одна з вершин має бути в покритті. Розгалужуємося на два випадки.

```
Function vertex_cover_search_tree(G, k):
    """
    Знаходить вершинне покриття розміру ≤ k
    G = (V, E) - граф як список ребер
    """
    if k < 0:
```

```
        return None  # Перевищили бюджет

    if len(G.edges) == 0:
        return set()  # Всі ребра покриті

    # Вибираємо довільне непокрите ребро
    (u, v) = G.edges[0]

    # Гілка 1: включаємо u в покриття
    G1 = remove_vertex(G, u)
    cover1 = vertex_cover_search_tree(G1, k-1)
    if cover1 is not None:
        return cover1.union({u})

    # Гілка 2: включаємо v в покриття
    G2 = remove_vertex(G, v)
    cover2 = vertex_cover_search_tree(G2, k-1)
    if cover2 is not None:
        return cover2.union({v})

    return None  # Покриття розміру ≤ k не існує
```

Аналіз складності:

- Глибина дерева: $\leq k$ (на кожному рівні зменшуємо параметр)

- Розгалуження: 2 на кожному вузлі

- Кількість листків: $\leq 2^k$

- Час на вузол: $O(n + m)$

- Загальний час: $O(2^k \cdot (n + m))$

Онлайн алгоритми та конкурентне відношення: математичний розгляд невизначеності

Вступ: гра проти всезнаючого супротивника

Уявіть, що ви граєте в шахи, але з особливими правилами: ваш супротивник бачить усі ваші майбутні ходи наперед, а ви мусите приймати рішення, не знаючи навіть його наступного ходу. Звучить несправедливо? Саме в таких умовах працюють онлайн алгоритми – вони приймають рішення послідовно, без знання майбутніх вхідних даних, змагаючись проти оптимального офлайн алгоритму, який знає всю послідовність наперед.

Ця галузь теоретичної інформатики, започаткована Слейтором та Тар'яном у 1985 році, не лише елегантна математично, але й надзвичайно практична. Від кешування веб-сторінок до біржової торгівлі – онлайн алгоритми всюди, де потрібно приймати рішення в умовах невизначеності.

Теоретичні основи: формалізація невизначеності

Визначення та основні концепції

Онлайн алгоритм – це алгоритм, який обробляє вхідні дані послідовно, приймаючи незворотні рішення для кожного елемента без знання майбутніх входів. На противагу йому, офлайн алгоритм має повну інформацію про всю вхідну послідовність заздалегідь.

Центральним поняттям аналізу є конкурентне відношення. Для задачі мінімізації, онлайн алгоритм ALG називається ρ-конкурентним, якщо існує константа b така, що для кожної вхідної послідовності σ:

$$\text{cost}_{\text{ALG}}(\sigma) \leq \rho \cdot \text{cost}_{\text{OPT}}(\sigma) + b$$

де $\text{cost}_{\text{OPT}}(\sigma)$ – вартість оптимального офлайн алгоритму. Коефіцієнт ρ і є конкурентним відношенням.

Парадигма супротивника

Аналіз конкурентності використовує модель супротивника: уявний супротивник генерує найгіршу можливу вхідну послідовність, максимізуючи відношення вартості онлайн алгоритму до оптимального розв'язку. Це створює своєрідну гру між алгоритмом та супротивником, де алгоритм намагається мінімізувати, а супротивник – максимізувати конкурентне відношення.

Розрізняють кілька типів супротивників: - Обмежений супротивник (oblivious adversary): не знає випадкових виборів рандомізованого алгоритму - Адаптивний супротивник: бачить поточний стан алгоритму та його випадкові вибори

Алгоритм LRU: елегантність простоти

Задача кешування сторінок

Розглянемо класичну задачу кешування сторінок (paging problem). У нас є швидка пам'ять (кеш) розміром k сторінок та повільна пам'ять з N сторінками ($k < N$). Коли запитується сторінка: - Якщо вона в кеші – це "попадання" з нульовою вартістю - Якщо її немає – це "промах" з вартістю 1, і потрібно вибрати сторінку для видалення

Алгоритм LRU (Least Recently Used) видаляє сторінку, до якої найдовше не було звернень. Інтуїтивно це розумно – якщо до сторінки давно не зверталися, ймовірно, вона не знадобиться найближчим часом.

Доведення k-конкурентності LRU

Теорема: LRU є k-конкурентним для задачі кешування.

Доведення використовує техніку розбиття на блоки. Візьмемо вхідну послідовність σ і розіб'ємо її на максимальні блоки $\sigma_1, \sigma_2, \dots, \sigma_b$, де кожен блок містить не більше k різних сторінок.

Крок 1: LRU робить не більше k промахів на блок. Чому? Тому що LRU промахується максимум один раз для кожної унікальної сторінки в блоці. Після того, як сторінка потрапила в кеш, LRU не видалить її, доки не буде запитано k інших різних сторінок – а це неможливо в межах одного блоку за визначенням.

Крок 2: Оптимальний алгоритм робить принаймні $(b-1)$ промахів. Розглянемо "зсунуті блоки": перший блок + перший запит другого блоку. Це $k+1$ різних сторінок, тому навіть оптимальний алгоритм мусить зробити хоча б один промах. За індукцією, на кожному переході між блоками відбувається принаймні один промах.

Висновок: Конкурентне відношення $= \frac{\text{LRU}(\sigma)}{\text{OPT}(\sigma)} \leq \frac{b \cdot k}{b-1} \to k$ при $b \to \infty$.

Краса цього доведення в тому, що воно показує: LRU досягає оптимального конкурентного відношення k серед усіх детерміністичних алгоритмів!

Обчислення конкурентних відношень: класичні приклади

Задача оренди лиж (Ski Rental Problem)

Ця задача – чудова ілюстрація дилеми "купувати чи орендувати". Оренда лиж коштує r грошей на день, покупка – p грошей (де $r \leq p$). Ви не знаєте, скільки днів катаєтеся.

Оптимальний детерміністичний алгоритм: Орендуйте $\lfloor p/r \rfloor - 1$ днів, потім купуйте.

Аналіз конкурентного відношення: - Якщо катаєтеся $t < p/r$ днів: обидва алгоритми платять $t \cdot r$ (відношення = 1) - Якщо катаєтеся $t \geq p/r$ днів: - Алгоритм платить: $(p/r - 1) \cdot r + p = 2p - r$ - Оптимальний платить: p - Відношення: $(2p - r)/p = 2 - r/p$

Отже, алгоритм є $(2 - r/p)$-конкурентним, що майже 2 для малих r/p.

Задача оновлення списку (List Update)

У цій задачі ми підтримуємо зв’язаний список елементів. Доступ до елемента на позиції i коштує i. Після доступу можна безкоштовно перемістити елемент вперед.

Алгоритм Move-to-Front (MTF): Після доступу переміщуємо елемент на початок списку.

Теорема: MTF є 2-конкурентним.

Доведення використовує метод потенціальної функції. Визначимо потенціал Ф як кількість інверсій між списком MTF та оптимальним списком (пара елементів у різному порядку).

Для запиту елемента x на позиції k в MTF та k^* в OPT: - Нехай v = кількість елементів перед x в MTF, але після x в OPT - Тоді $k \leq k^* + v$ - Амортизована вартість MTF = $k + \Delta\Phi = k + (k - 1 - 2v) \leq 2k^* - 1$

Сумуючи по всіх запитах, отримуємо 2-конкурентність.

Задача k-серверів

Це узагальнення багатьох онлайн задач. Маємо k серверів у метричному просторі. Запити надходять у точки простору, і ми мусимо перемістити сервер до кожного запиту. Мета – мінімізувати загальну пройдену відстань.

Гіпотеза k-серверів: Існує k-конкурентний алгоритм для будь-якого метричного простору.

Відомі результати: - Для $k = 2$: гіпотеза доведена - Загальний випадок: найкращий відомий алгоритм (Work Function Algorithm) є $(2k - 1)$-конкурентним - Для дерев: існує оптимальний k-конкурентний алгоритм

Задача кешування – це спеціальний випадок k-серверів у рівномірній метриці (всі відстані = 1).

Методи доведення нижніх меж

Принцип мінімаксу Яо

Це потужний інструмент для доведення нижніх меж рандомізованих алгоритмів. Принцип встановлює еквівалентність між: - Продуктивністю найкращого рандомізованого алгоритму проти найгіршого входу - Продуктивністю найкращого детерміністичного алгоритму проти найгіршого розподілу входів

Математична формулювання:

$$\min_{A \in \mathcal{A}} \mathbb{E}[C(I_p, A)] \leq \max_{I \in \mathcal{I}} \mathbb{E}[C(I, A_q)]$$

де p – розподіл над входами, q – розподіл над алгоритмами.

Застосування до кешування

Використовуючи принцип Яо, можна довести, що будь-який рандомізований алгоритм кешування має конкурентне відношення принаймні $H_k \approx \ln k$ проти обмеженого супротивника.

Ідея доведення: Розглядаємо рівномірний розподіл на послідовностях, що циклічно проходять через $k+1$ сторінок. Будь-який детерміністичний алгоритм буде поводитися погано в середньому на цьому розподілі.

Конструкції супротивника

Для детерміністичних алгоритмів супротивник може передбачити всі майбутні кроки алгоритму. Класична конструкція для кешування: 1. Супротивник підтримує $k+1$ “активних” сторінок 2. Завжди запитує сторінку, якої немає в кеші алгоритму 3. Це гарантує промах на кожному кроці для онлайн алгоритму 4. Оптимальний офлайн алгоритм може тримати в кеші всі крім однієї сторінки

Рандомізовані онлайн алгоритми

Переваги рандомізації

Рандомізація дозволяє “заплутати” супротивника, не даючи йому експлуатувати передбачувану поведінку. Для рандомізованого алгоритму R визначаємо очікуване конкурентне відношення:

$$\mathbb{E}[R(\sigma)] \leq c \cdot \mathrm{OPT}(\sigma) + b$$

де очікування береться за випадковими виборами алгоритму.

Алгоритм маркування для кешування

Цей елегантний рандомізований алгоритм досягає оптимального конкурентного відношення $2H_k$:

1. Підтримуємо біт-маркер для кожної сторінки в кеші

2. При попаданні: встановлюємо маркер

3. При промаху: видаляємо випадкову немарковану сторінку, завантажуємо нову, маркуємо її

4. Коли всі сторінки марковані: знімаємо всі маркери

Доведення використовує складний амортизований аналіз, порівнюючи фази алгоритму з оптимальним розв’язком.

Рандомізована оренда лиж

Оптимальний рандомізований алгоритм вибирає день покупки i з ймовірністю:

$$p_i = \frac{(p-1)^{p-i}}{p(1-(1-1/p)^p)} \cdot \frac{1}{p}$$

Це дає конкурентне відношення $\frac{e}{e-1} \approx 1.58$ – значно краще за детерміністичне 2!

Онлайн алгоритми як ігри

Конкурентний аналіз природно вписується в теорію ігор як гра з нульовою сумою: - Гравець 1 (Алгоритм): вибирає стратегію обробки запитів - Гравець 2 (Супротивник): вибирає вхідну послідовність - Виграш: конкурентне відношення (алгоритм мінімізує, супротивник максимізує)

Прямо-двоїстий метод

Цей потужний метод походить з лінійного програмування та теорії ігор:

1. Формулюємо онлайн задачу як лінійну програму

2. Підтримуємо допустимі прямий та двоїстий розв'язки онлайн

3. Оновлюємо обидва розв'язки при надходженні запитів

4. Конкурентне відношення пов'язане з відношенням цільових функцій

Приклад: Для задачі оренди лиж двоїсті змінні відстежують "терміновість" покупки проти продовження оренди.

Навчання з підкріпленням та регрет

Сучасні дослідження пов'язують онлайн алгоритми з машинним навчанням через концепцію регрету – різниці між продуктивністю алгоритму та найкращої стратегії заднім числом:

$$\text{Regret}_T = \sum_{t=1}^{T} f_t(x_t) - \min_{x^*} \sum_{t=1}^{T} f_t(x^*)$$

Алгоритми типу Multiplicative Weights досягають регрету $O(\sqrt{T \log n})$, що сублінійно за T.

Теорія онлайн алгоритмів демонструє, як математична елегантність поєднується з практичною необхідністю. Від простого правила LRU, яке виявляється оптимальним, до складних рандомізованих стратегій – ця галузь показує, що навіть в умовах повної невизначеності можна давати сильні гарантії продуктивності.

Конкурентний аналіз не просто песимістична оцінка найгіршого випадку – це спосіб зрозуміти фундаментальні обмеження алгоритмів в умовах неповної інформації. Зв’язки з теорією ігор, машинним навчанням та оптимізацією роблять цю галузь живою та актуальною.

Майбутнє онлайн алгоритмів лежить на перетині теорії та практики: використання машинного навчання для покращення передбачень, адаптивні алгоритми для змінних середовищ, та нові моделі, що виходять за межі класичного конкурентного аналізу. Але основна ідея залишається незмінною – як приймати хороші рішення, коли майбутнє невідоме?

Розділ 17: Квантові обчислення чи плутанина?

Настал час перейти до найбільш дивовижного та контрінтуїтивного розділу нашого навчання алгоритмів. Сама назва, **“Квантові обчислення чи плутанина?”**, відображає подвійне відчуття, яке викликає ця тема: з одного боку — захоплення неймовірними можливостями нової обчислювальної парадигми, а з іншого — подив перед дивними й парадоксальними законами квантового світу.

Мета цього розділу — перетворити початкову плутанину на ясне розуміння, показавши, що за екзотичними явищами, як-от суперпозиція та квантова заплутаність, стоїть строга математична логіка, здатна вирішувати задачі, недоступні для найпотужніших сучасних суперкомп’ютерів.

Ми почнемо з руйнування звичної нам класичної інтуїції, познайомимося з кубітом — фундаментальною одиницею квантової інформації, яка може бути одночасно і 0, і 1. Далі ми дослідимо, як ця дивна властивість дозволяє створювати революційні алгоритми, такі як алгоритм Шора, здатний зламати більшість сучасних криптографічних систем, та алгоритм Гровера, що обіцяє кардинально прискорити пошук.

Але, розкриваючи квантову загрозу, ми одразу ж розглянемо і відповідь на неї. Ми зануримося у світ постквантової криптографії — нового покоління класичних алгоритмів, розроблених для того, щоб протистояти атакам як сьогоднішніх, так і майбутніх квантових комп’ютерів.

Цей розділ — це не просто огляд екзотичних алгоритмів, а погляд у майбутнє обчислень та цифрової безпеки, де закони фізики диктують нові правила гри.

Вступ: Чому квантовий світ "заплутаний"?

Слово “заплутаний” у назві цього розділу використано невипадково. Воно одночасно передає два ключові аспекти: по-перше, відчуття збентеження та плутанини, яке виникає у кожного, хто вперше стикається з дивовижними законами квантової механіки. По-друге, воно є прямим натяком на один з найглибших і найпотужніших феноменів цього світу — квантову заплутаність.

Наша мета — провести вас крізь цю початкову плутанину, показавши, що за нею стоїть строга логіка, яка відкриває двері до абсолютно нової ери обчислень.

Щоб зрозуміти, звідки береться ця “дивність”, нам потрібно відмовитися від звичної класичної інтуїції та познайомитися з двома фундаментальними принципами, на яких тримається вся квантова магія.

Суперпозиція: більше ніж “або-або”

У світі, який ми бачимо, об’єкт може перебувати лише в одному стані в один момент часу. Класичний біт інформації — це або 0, або 1. Вимикач — увімкнений або вимкнений. Монета, що лежить на столі, — “орел” або “решка”.

Квантовий світ працює за принципом “і-і”. Фундаментальна одиниця квантової інформації, кубіт, може одночасно бути і 0, і 1, і нескінченною кількістю станів між ними. Це явище називається суперпозицією.

Уявіть собі монету, що обертається в повітрі. Доки вона не впала, вона не є ані “орлом”, ані “решкою” — вона перебуває в невизначеному стані, що містить обидві можливості. Лише в момент вимірювання (коли монета падає на стіл) вона приймає один конкретний класичний стан. Так само і кубіт: до вимірювання він існує в суперпозиції всіх своїх можливих станів, а вимірювання змушує його “обрати” один з них, причому результат цього вибору є імовірнісним.

Саме ця здатність кубіта зберігати значно більше інформації, ніж класичний біт, є першим джерелом потужності квантових комп’ютерів.

Квантова заплутаність: “моторошна дія на відстані”

Якщо суперпозиція здається дивною, то квантова заплутаність межує з науковою фантастикою. Це явище, яке Альберт Айнштайн назвав “моторошною дією на відстані”, описує глибокий та загадковий зв’язок між двома або більше кубітами.

Уявіть, що ми створили пару “заплутаних” кубітів. Їхня особливість у тому, що їхні долі назавжди пов’язані. Навіть якщо рознести ці кубіти на протилежні кінці галактики, вони залишатимуться єдиною системою. Якщо ми виміряємо стан одного кубіта і отримаємо, наприклад, 0, ми миттєво будемо знати, що вимірювання другого кубіта дасть результат 1 (або навпаки, залежно від типу заплутаності).

Ключові моменти заплутаності:

- Миттєва кореляція: Результати вимірювань заплутаних частинок скорельовані, і цей зв’язок проявляється миттєво, незалежно від відстані.

- Невизначеність до вимірювання: До моменту вимірювання стан кожного окремого кубіта є невизначеним. Лише спільний стан системи є визначеним.

Цей феномен не передає інформацію швидше за світло, але він створює потужні кореляції, яких не існує в класичному світі. Саме квантова заплутаність дозволяє виконувати складні операції одночасно над багатьма кубітами, що є основою для таких алгоритмів, як алгоритм Шора.

Суперпозиція та заплутаність є двома стовпами, на яких стоїть уся “плутанина” квантового світу. Водночас вони є найпотужнішим ресурсом, який квантові комп’ютери використовують для досягнення своєї неймовірної обчислювальної переваги. У наступних підрозділах ми побачимо, як ці фізичні явища формалізуються мовою математики та втілюються в реальних квантових алгоритмах.

Криза класичної інтуїції

Класична інтуїція народжується з повсякденного світу тіл, що мають визначені властивості, рухаються по неперервних траєкторіях і не впливають одне на одного на відстані. Ця інтуїція спирається на кілька непомітних припущень:

- властивості об’єкта існують незалежно від акту спостереження

- будь-які дві властивості можна виміряти одночасно й приписати об’єкту спільний набір значень

- логічні висловлювання про систему утворюють звичайну булеву алгебру

- ймовірності підпорядковуються класичному правилу додавання для взаємовиключних подій

- віддалені системи мають лише локальні кореляції

На квантовому рівні кожне з цих припущень дає збій. Нижче — карта головних «зламів» із конкретними прикладами та поясненнями, що саме виявляється хибним у класичній картині.

1. Суперпозиція і принцип амплітуд: порушення класичного правила повної ймовірності

У класичній теорії, якщо події взаємовиключні, то ймовірності додаються. У двощілинному експерименті це спокусливо записати як

$$P(\text{потрапити в точку A}) \stackrel{?}{=} P(A \mid \text{ліва щілина}) + P(A \mid \text{права щілина}).$$

Але експеримент дає інтерференційну картину, і ця рівність не виконується. Причина не у «дивній» ймовірності, а в іншому об'єкті обчислень: у квантовій механіці додаються не ймовірності, а комплексні амплітуди, і вже **після** додавання береться квадрат модуля. Коли щілини відкриті одночасно, «події» ще не стали взаємовиключними — система перебуває в суперпозиції шляхів, і саме ця суперпозиція породжує додатковий інтерференційний доданок. Висновок: класичне правило додавання ймовірностей працює лише для подій, які справді взаємовиключні внаслідок вимірювання або декогеренції.

2. Невизначеність через некомутативність: неможливість спільних точних значень

Класика передбачає, що можна одночасно знати «координату й імпульс» із довільною точністю. У квантовій механіці оператори координати $\hat{x}$ і імпульсу $\hat{p}$ не комутують:

$$[\hat{x}, \hat{p}] = i\hbar.$$

Звідси випливає співвідношення невизначеностей

$$\sigma_x \sigma_p \geq \frac{\hbar}{2}.$$

Це не про технічні обмеження приладу, а про структуру фізичної теорії: для некомутуючих спостережних не існує стану з одночасно гострими значеннями. Тому класична ідея траєкторії «точка в кожен момент має і x, і p» непридатна на атомних масштабах.

3. Квантова логіка: збій дистрибутивності для «висловлювань» про систему

У класичній логіці виконується

$$P \wedge (Q \vee R) = (P \wedge Q) \ \vee \ (P \wedge R).$$

У квантовій логіці, де висловлюванням відповідають проєктори на підпростори Гільбертового простору, дистрибутивність взагалі не є тотожно істинною. Приклад зі спіном:

- нехай P — «спін уздовж z додатний»

- Q — «спін уздовж x додатний»

- R — «спін уздовж x від'ємний»

Класично $Q \vee R$ — це «істина», але в квантовій логіці $P \wedge (Q \vee R)$ не збігається з $(P \wedge Q) \vee (P \wedge R)$, бо підпростори для x та z — несумісні: відповідні

проєктори не комутують. Звідси мораль: булеві закони для висловлювань про несумісні вимірювання застосовувати не можна.

4. Контекстуальність: значення спостережної залежать від набору одночасно вимірюваних величин

Класика припускає несуперечливе «приховане» значення кожної фізичної величини, яке просто зчитується вимірюванням. Теорема Кохена—Спекера показує: для систем вимірності три й більше не існує глобального присвоєння значень, сумісного з усіма функціональними зв'язками, незалежно від контексту вимірювання. Інтуїтивно: відповідь приладу на «те саме запитання» може залежати від того, які ще сумісні спостережні ви міряєте одночасно. Таким чином, вимірювання не просто виявляє значення — воно частково його визначає.

5. Ентанглемент і позалокальні кореляції: крах комбінації локальності та реалізму

Класичне мислення очікує, що далеким системам притаманні лише локальні кореляції, і що можна відтворити статистику вимірювань через приховані параметри, не вдаючись до «миттєвих впливів». Нерівності Белла (зокрема CHSH) формалізують ці очікування. Експерименти на ентангльованих парах систем порушують ці нерівності: вимірювання дають кореляції, сильніші за будь-яку локально-реалістичну модель. Висновок: принаймні одне з класичних переконань — локальність або наївний реалізм — несумісне з даними.

6. Колапс, проєкція та декогеренція: чому «результат» не існує до вимірювання, але класика все ж виникає

Рівняння Шредінгера лінійне, тому система + прилад переходять у суперпозицію «усіх можливих показів». Як же ми бачимо один результат?

- проєкційний постулат каже, що стан скаче в один із власних станів оператора вимірювання з імовірностями за правилом Борна

- декогеренція пояснює, як взаємодія з середовищем руйнує когерентність між макроскопічно різними гілками, швидко діагоналізуючи матрицю густини у певній базі «вказівників»

Це не «класична межа» в сенсі точкових траєкторій, але практично знищує інтерференцію між макрогиілками, завдяки чому булева логіка та класична ймовірність стають дуже добрими апроксимаціями для спостережуваного світу.

7. Безперервні траєкторії та причинність: як саме змінюється мова опису

Класичні траєкторії — це наслідок одночасної визначеності x і p. У квантовій теорії еволюція стану детермінована (Шредінгер), але спостережні результати народжуються стохастично відповідно до амплітуд. Тому причинність не зникає — змінюється рівень опису: детермінізм існує для векторів стану, а не для окремих вимірювань. Спроби «повернути» класичну картину через приховані змінні неминуче стикаються з контекстуальністю або порушенням нерівностей Белла.

8. Коли класика знову працює: граничні режими

Класичні закони — не «хибні», а граничні. Вони повертаються, коли $\hbar$ ефективно малий щодо дій у задачі, коли суперпозиції сильно декогерують, а всі релевантні спостережні майже комутують. Тоді з'являється можливість одночасно наближено приписати значення багатьом величинам і повернути булеву логіку подій.

9. Що ми втрачаємо і що отримуємо.

- Замість додавання ймовірностей — додавання амплітуд і квадрат модуля.

- Замість одночасних точних значень — співвідношення невизначеностей і некомутативні оператори.

- Замість булевої логіки для всіх висловлювань — квантова логіка для несумісних вимірювань.

- Замість локально-реалістичних кореляцій — ентанглемент і порушення нерівностей Белла.

- Замість «вимірювання як пасивного читання» — контекстуальність і роль декогеренції у появі класичного результату.

Це і є зміст «кризи класичної інтуїції»: не те, що світ «нелогічний», а те, що логіка й імовірність класики застосовні лише в певних режимах. На квантовому рівні правильними об'єктами стають амплітуди, оператори, проєктори та щільності — і саме їхня математика диктує межі нашої інтуїції.

Два стовпи "плутанини": ключові відмінності квантового світу

Уявіть, що ви завжди жили у світі, де всі предмети поводяться передбачувано: м'яч летить по параболі, вода тече вниз, а монета може бути

або орлом, або решкою. Тепер вам кажуть, що на рівні атомів діють зовсім інші правила. Монета може бути одночасно і орлом, і решкою. Більше того, дві монети можуть бути таємничо пов'язані: варто подивитися на одну, як друга миттєво "дізнається" про це, навіть якщо знаходиться на іншому кінці Всесвіту.

Суперпозиція

Звучить як фантастика? Проте це реальність квантового світу. Давайте розберемося з двома найдивнішими явищами квантової фізики — **суперпозицією** та **квантовою заплутаністю**.

Суперпозиція: мистецтво бути в кількох місцях одночасно. Від звичайної монети до квантової

Візьмемо просту монету. У нашому повсякденному світі вона завжди має конкретний стан:

- Лежить орлом догори

- Лежить решкою догори

- Обертається в повітрі (але в кожен момент часу все одно орієнтована певним чином)

Навіть коли монета обертається так швидко, що ми не бачимо її сторін, вона все одно в кожну мить має конкретну орієнтацію. Ми просто не знаємо яку. Не знаємо, поки не подивимось.

Тепер уявімо "квантову монету" — наприклад, електрон, який може мати спін "вгору" або "вниз" (це щось на кшталт обертання, але в квантовому світі). Виявляється, електрон може перебувати в стані, коли він справді має спін і вгору, і вниз одночасно!

У квантовому світі стан об'єкта описується не просто словами "тут" або "там", "вгору" або "вниз". Замість цього використовується математичний об'єкт, який називається хвильовою функцією.

Давайте познайомимося з квантовими позначеннями поступово:

Крок 1: Позначення станів

- $|\uparrow\rangle$ — читається "кет-стрілка вгору", означає стан "спін вгору"

- $|\downarrow\rangle$ — читається "кет-стрілка вниз", означає стан "спін вниз"

Дивні дужки $|\rangle$ — це просто спосіб позначити квантовий стан. Думайте про них як про "контейнер" для стану.

Крок 2: Суперпозиція Квантова частинка може бути в комбінації станів:

$|\psi\rangle = a|\uparrow\rangle + b|\downarrow\rangle$

Читається: “псі дорівнює a помножити на стан вгору плюс b помножити на стан вниз”.

Тут:

- $|\psi\rangle$ (грецька буква “псі”) — повний стан частинки,

- a та b — числа, які показують “наскільки сильно” частинка перебуває в кожному стані,

- Знак “+” означає, що частинка перебуває в обох станах одночасно.

Математично - це лінійна комбінація векторів.

Крок 3: Що означають числа a та b?

Ці числа називаються амплітудами ймовірності. Коли ми вимірюємо частинку:

- Ймовірність знайти спін вгору = $|a|^2$ (модуль a в квадраті)

- Ймовірність знайти спін вниз = $|b|^2$ (модуль b в квадраті)

Важливе правило: $|a|^2 + |b|^2 = 1$ (сума ймовірностей завжди дорівнює 1, або 100%)

Розглянемо електрон в стані $|\psi\rangle$:

$|\psi\rangle = (1/\sqrt{2})|\uparrow\rangle + (1/\sqrt{2})|\downarrow\rangle$

Що це означає?

- Амплітуда для “вгору”: $a = 1/\sqrt{2} \approx 0.707$

- Амплітуда для “вниз”: $b = 1/\sqrt{2} \approx 0.707$

- Ймовірність виміряти “вгору”: $|a|^2 = 1/2 = 50\%$

- Ймовірність виміряти “вниз”: $|b|^2 = 1/2 = 50\%$

Але до вимірювання електрон дійсно перебуває в обох станах!

Реальні приклади суперпозиції

1. Електрон в атомі В атомі водню електрон не обертається по колу, як планета. Він існує в “хмарі” — суперпозиції всіх можливих положень. Ця хмара має певну форму (сферичну, гантелеподібну тощо), яка визначає хімічні властивості атома.

2. Світло проходить крізь щілини Коли фотон (частинка світла) проходить через дві щілини, він проходить через обидві одночасно! Це створює інтерференційну картину на екрані — смуги світла і темряви.

3. Молекули під час хімічних реакцій Деякі молекули можуть одночасно мати різні форми. Наприклад, молекула аміаку (NH_3) може бути в суперпозиції двох дзеркальних конфігурацій.

Чому ми не бачимо суперпозицію навколо себе?

Кіт Шредінгера — це уявний експеримент, де кіт нібито перебуває в суперпозиції живого і мертвого станів. Чому ж реальні коти не можуть бути "наполовину живими"?

Відповідь: **декогеренція**. Великі об'єкти постійно взаємодіють з навколишнім середовищем:

- Молекули повітря стикаються з об'єктом

- Фотони світла відбиваються від поверхні

- Теплове випромінювання йде в усі боки

Кожна така взаємодія — це своєрідне "вимірювання", яке руйнує суперпозицію. Чим більший об'єкт, тим швидше це відбувається:

- Електрон в ідеальному вакуумі: може зберігати суперпозицію годинами

- Молекула: мільйонні частки секунди

- Пилинка: трильйонні частки секунди

- Кіт: настільки швидко, що це неможливо виміряти

Фундаментальні поняття

Кубіт

1. Обмеження класичних бітів

У класичній інформатиці вся інформація кодується за допомогою бітів - найменших одиниць інформації, які можуть перебувати в одному з двох станів: 0 або 1. Ця бінарна система лежить в основі всіх сучасних комп'ютерів, від смартфонів до суперкомп'ютерів. Кожен транзистор у процесорі фактично реалізує біт, перебуваючи або у "ввімкненому" (1), або у "вимкненому" (0) стані.

Однак природа на фундаментальному рівні підкоряється законам квантової механіки, які дозволяють існування станів, неможливих у класичному світі. Квантові системи можуть перебувати в суперпозиції - одночасно в кількох станах до моменту вимірювання. Саме це властивість лежить в основі квантових обчислень і відкриває принципово нові можливості для обробки інформації.

2. Народження ідеї квантових обчислень

Ідея використання квантових систем для обчислень виникла на початку 1980-х років. Річард Фейнман у своїй знаменитій лекції 1981 року зазначив, що для ефективного моделювання квантових систем потрібні квантові комп'ютери. Він підкреслив, що класичні комп'ютери потребують експоненційних ресурсів для симуляції навіть простих квантових систем.

Пол Беніофф у 1980 році показав теоретичну можливість побудови квантової машини Тюрінга, а Девід Дойч у 1985 році формалізував концепцію універсального квантового комп'ютера. Але справжній прорив стався в 1994 році, коли Пітер Шор продемонстрував квантовий алгоритм факторизації великих чисел, який працює експоненційно швидше за найкращі відомі класичні алгоритми.

2. Фізичні основи кубіта

2.1. Квантові системи як носії інформації

Кубіт (квантовий біт, від англ. quantum bit) - це квантова система з двома базовими станами, яку можна використовувати для зберігання та обробки квантової інформації. На відміну від класичного біта, кубіт може перебувати не лише в станах $|0\rangle$ або $|1\rangle$, але й у будь-якій їх квантовій суперпозиції.

Фізично кубіт може бути реалізований різними способами:

1. Спін електрона: Електрон має внутрішній кутовий момент - спін, який може бути орієнтований "вгору" ($|\uparrow\rangle = |0\rangle$) або "вниз" ($|\downarrow\rangle = |1\rangle$) відносно заданої осі.

2. Поляризація фотона: Фотон може мати горизонтальну ($|H\rangle = |0\rangle$) або вертикальну ($|V\rangle = |1\rangle$) поляризацію.

3. Енергетичні рівні атома: Атом може перебувати в основному ($|g\rangle = |0\rangle$) або збудженому ($|e\rangle = |1\rangle$) стані.

4. Надпровідні контури: Струм у надпровідному кільці може текти за годинниковою стрілкою ($|0\rangle$) або проти ($|1\rangle$).

5. Іонні пастки: Іони, утримувані електромагнітними полями, можуть мати різні енергетичні стани.

2.2. Математичний опис стану кубіта

Загальний стан кубіта описується як лінійна комбінація базових станів:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

де α та β - комплексні числа, які називаються амплітудами ймовірності. Ці коефіцієнти задовольняють умову нормування:

$$|\alpha|^2 + |\beta|^2 = 1$$

Ця умова гарантує, що сумарна ймовірність знайти кубіт в одному зі станів дорівнює 1.

Приклад: Розглянемо кубіт у стані:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

При вимірюванні цього кубіта ми отримаємо:

- Стан $|0\rangle$ з ймовірністю $|\frac{1}{\sqrt{2}}|^2 = \frac{1}{2}$
- Стан $|1\rangle$ з ймовірністю $|\frac{1}{\sqrt{2}}|^2 = \frac{1}{2}$

Цей стан називається станом рівної суперпозиції і позначається як $|+\rangle$.

2.3. Нотація Дірака

У квантовій механіці використовується спеціальна нотація, введена Полом Діраком:

- Кет-вектор $|\psi\rangle$ представляє квантовий стан
- Бра-вектор $\langle\psi|$ - це комплексно спряжений транспонований вектор
- Скалярний добуток $\langle\varphi|\psi\rangle$ дає амплітуду переходу між станами
- Зовнішній добуток $|\psi\rangle\langle\varphi|$ створює оператор

Для кубіта базові стани можна записати як вектори:

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, \quad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$$

Тоді загальний стан:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix}\alpha\\\beta\end{pmatrix}$$

Скалярний добуток $\langle\phi|\psi\rangle$ дає нам амплітуду переходу між станами — число, що показує, наскільки два стани “схожі”:

$$\langle \phi|\psi\rangle = \phi_0^*\psi_0 + \phi_1^*\psi_1$$

Для ортогональних станів (як $|0\rangle$ та $|1\rangle$) скалярний добуток дорівнює нулю:

$$\langle 0|0\rangle = 1, \quad \langle 0|1\rangle = 0, \quad \langle 1|1\rangle = 1$$

3. Ключові відмінності кубіта від класичного біта

3.1. Суперпозиція

Класичний біт: Може бути тільки 0 АБО 1. Це дискретні, взаємовиключні стани.

Кубіт: Може бути в суперпозиції 0 ТА 1 одночасно. Математично:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Це не означає, що кубіт "частково 0 і частково 1". Він дійсно перебуває в обох станах одночасно до моменту вимірювання.

Візуальна аналогія: Уявіть монету, що обертається в повітрі. Класичний біт - це монета, що лежить на столі (орел або решка). Кубіт - це монета в польоті, яка одночасно є і орлом, і решкою.

3.2. Вимірювання та колапс

Вимірювання — це унікальний процес у квантовій механіці, який кардинально відрізняється від будь-якої іншої операції. Коли ми вимірюємо квантову систему, відбувається колапс хвильової функції — суперпозиція зникає, і система переходить в один з базисних станів.

Постулат вимірювання стверджує, що для стану $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$:

Ймовірність отримати результат 0:

$$P(0) = |\alpha|^2 = |\langle 0|\psi\rangle|^2$$

Ймовірність отримати результат 1:

$$P(1) = |\beta|^2 = |\langle 1|\psi\rangle|^2$$

Важливо розуміти: після вимірювання стан системи змінюється! Якщо ми отримали результат 0, стан стає $|0\rangle$, незалежно від того, яким він був до вимірювання. Це незворотній процес — інформація про початкові значення α та β втрачається назавжди.

Приклад вимірювання: Розглянемо стан $|\psi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$

Ймовірності:

$$P(0) = \left(\frac{\sqrt{3}}{2}\right)^2 = \frac{3}{4} = 75\%$$

$$P(1) = \left(\frac{1}{2}\right)^2 = \frac{1}{4} = 25\%$$

Якщо провести 1000 вимірювань ідентично підготовлених станів, ми отримаємо приблизно 750 нулів та 250 одиниць.

3.3. Неможливість копіювання

Класичний біт: Можна легко скопіювати: якщо x = 0, то створити y = 0.

Кубіт: Теорема про заборону клонування стверджує, що неможливо створити копію невідомого квантового стану. Не існує квантової операції, яка б реалізувала:

$$|\psi\rangle|0\rangle \rightarrow |\psi\rangle|\psi\rangle$$

для довільного стану |ψ⟩.

Це має глибокі наслідки для квантової інформації та криптографії.

3.4. Континуум станів

Класичний біт: Має лише 2 можливі стани.

Кубіт: Має нескінченну кількість можливих станів (всі можливі суперпозиції). Але при вимірюванні все одно отримаємо лише 0 або 1.

3.5. Інтерференція

Класичний біт: Біти не інтерферують між собою.

Кубіт: Квантові амплітуди можуть інтерферувати конструктивно або деструктивно, що є основою квантових алгоритмів.

4. Сфера Блоха: Геометрична візуалізація кубіта

4.1. Мотивація та визначення

Оскільки стан кубіта визначається двома комплексними числами з умовою нормування, природно виникає питання: чи можна візуалізувати всі можливі стани кубіта? Відповідь - так, і для цього використовується сфера Блоха.

Сфера Блоха - це одинична сфера в тривимірному просторі, де кожна точка на поверхні представляє чистий стан кубіта, а точки всередині - змішані стани.

4.2. Побудова сфери Блоха

Почнемо з загального стану кубіта:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Використовуючи умову нормування $|\alpha|^2 + |\beta|^2 = 1$ та той факт, що глобальна фаза не впливає на фізичні властивості, можемо параметризувати стан як:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle$$

де: - $\theta \in [0, \pi]$ - полярний кут (кут від осі z) - $\phi \in [0, 2\pi)$ - азимутальний кут (кут в площині xy)

4.3. Координати на сфері Блоха

Кожному стану кубіта відповідає точка на сфері з декартовими координатами:

$$x = \sin\theta\cos\phi$$
$$y = \sin\theta\sin\phi$$
$$z = \cos\theta$$

Ці координати пов'язані з очікуваними значеннями матриць Паулі:

$$\langle\sigma_x\rangle = \langle\psi|\sigma_x|\psi\rangle = x$$
$$\langle\sigma_y\rangle = \langle\psi|\sigma_y|\psi\rangle = y$$
$$\langle\sigma_z\rangle = \langle\psi|\sigma_z|\psi\rangle = z$$

4.4. Особливі точки на сфері Блоха

Полюси: - Північний полюс (0, 0, 1): стан $|0\rangle$ - Південний полюс (0, 0, -1): стан $|1\rangle$

Екватор (z = 0):

- (1, 0, 0): стан $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$

- (-1, 0, 0): стан $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

- (0, 1, 0): стан $|+i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$

- (0, -1, 0): стан $|-i\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$

4.5. Візуалізація операцій на сфері Блоха

Обертання навколо осей:

- Обертання навколо осі z на кут α: зміна фази між $|0\rangle$ та $|1\rangle$

- Обертання навколо осі x на кут β: перемішування $|0\rangle$ та $|1\rangle$ з дійсними коефіцієнтами

- Обертання навколо осі y на кут γ: перемішування $|0\rangle$ та $|1\rangle$ з уявними коефіцієнтами

Квантові гейти як обертання:

- Гейт Паулі-X: обертання на π навколо осі x
- Гейт Паулі-Y: обертання на π навколо осі y
- Гейт Паулі-Z: обертання на π навколо осі z
- Гейт Адамара: обертання на π навколо осі $(x+z)/\sqrt{2}$

4.6. Переваги візуалізації через сферу Блоха

1. Інтуїтивність: Складні квантові операції стають простими геометричними обертаннями.

2. Наочність суперпозиції: Стани суперпозиції знаходяться між полюсами, що візуально показує їх "проміжний" характер.

3. Унітарність операцій: Всі квантові операції над одним кубітом - це обертання сфери, що зберігають її форму.

4. Ортогональність станів: Ортогональні квантові стани відповідають антиподальним точкам на сфері.

4.7. Обмеження сфери Блоха

Важливо розуміти обмеження цієї візуалізації:

1. Тільки для одного кубіта: Для систем з кількома кубітами потрібні складніші представлення.

2. Не показує глобальну фазу: Стани $|\psi\rangle$ та $e^{i\alpha}|\psi\rangle$ представлені однією точкою.

3. Змішані стани: Знаходяться всередині сфери, що ускладнює інтерпретацію.

8. Заплутаність - ключ до квантової переваги

Квантова заплутаність — це явище, яке не має аналогів у класичному світі. Коли дві або більше частинок заплутані, вони утворюють єдину квантову систему, стан якої не можна описати як простий добуток станів окремих частинок. Це означає, що частинки втрачають свою індивідуальність і стають частинами чогось більшого.

Найпростіший приклад — стан Белла:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Цей стан говорить нам: "Обидві частинки або в стані 0, або обидві в стані 1, але ми не знаємо в якому саме, поки не виміряємо". Це не просто брак

інформації з нашого боку — самі частинки “не знають” свого стану до вимірювання!

Парадоксальні властивості заплутаності здивували навіть творців квантової механіки. Якщо виміряти одну частинку заплутаної пари і отримати 0, друга миттєво “дізнається” про це і також стане 0, незалежно від відстані між ними. Ейнштейн називав це “моторошною дією на відстані” і вважав доказом неповноти квантової механіки. Однак експерименти підтвердили: заплутаність реальна.

Чому це не порушує теорію відносності? Хоча кореляція між частинками встановлюється миттєво, передати інформацію через заплутаність неможливо. Результати вимірювань випадкові — ми не можемо контролювати, що отримаємо (0 чи 1). Тільки порівнявши результати (що вимагає класичного каналу зв’язку), ми побачимо кореляцію.

Чотири стани Белла

Існує рівно чотири максимально заплутані стани двох кубітів, які утворюють ортонормований базис для двокубітного простору. Ці стани називаються станами Белла на честь Джона Белла, який довів їх фундаментальну роль у квантовій механіці.

Стан $|\Phi^+\rangle$ — симетричний стан з однаковими значеннями:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Якщо перша частинка 0, друга теж 0. Якщо перша 1, друга теж 1.

Стан $|\Phi^-\rangle$ — симетричний стан з протилежною фазою:

$$|\Phi^-\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}$$

Кореляція та сама, але з від’ємною фазою для стану $|11\rangle$.

Стан $|\Psi^+\rangle$ — антисиметричний стан:

$$|\Psi^+\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}$$

Частинки завжди в протилежних станах: якщо одна 0, інша 1.

Стан $|\Psi^-\rangle$ — антисиметричний стан з протилежною фазою:

$$|\Psi^-\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}$$

Також протилежні значення, але з від’ємною фазою.

Математична елегантність станів Белла полягає в тому, що вони утворюють повний базис — будь-який стан двох кубітів можна розкласти по станах Белла. Це критично для квантової телепортації, де ми використовуємо вимірювання в базисі Белла.

Створення заплутаності

Заплутані стани не існують у природі в готовому вигляді — їх потрібно створювати. Найпростіший спосіб створити стан Белла $|\Phi^+\rangle$ використовує лише два квантові гейти.

Алгоритм створення:

Крок 1: Початковий стан Починаємо з двох кубітів у стані $|00\rangle$:

$$|\psi_0\rangle = |00\rangle = |0\rangle \otimes |0\rangle$$

Крок 2: Створення суперпозиції Застосовуємо гейт Адамара до першого кубіта:

$$|\psi_1\rangle = H_1|00\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |0\rangle = \frac{|00\rangle + |10\rangle}{\sqrt{2}}$$

Тепер перший кубіт у суперпозиції, а другий все ще в стані $|0\rangle$.

Крок 3: Створення заплутаності Застосовуємо CNOT з першим кубітом як контрольним:

$$|\psi_2\rangle = CNOT_{12}\frac{|00\rangle + |10\rangle}{\sqrt{2}}$$

CNOT діє так: - $|00\rangle \to |00\rangle$ (перший 0, другий не змінюється) - $|10\rangle \to |11\rangle$ (перший 1, другий перевертається)

Результат:

$$|\psi_2\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\Phi^+\rangle$$

Фізична реалізація залежить від типу кубітів. Для фотонів використовують нелінійні кристали, які при поглинанні одного фотона випромінюють два заплутані. Для атомів використовують лазерні імпульси. Для надпровідних кубітів — мікрохвильові імпульси.

Квантові гейти: Аналоги логічних вентилів для маніпуляції кубітами

У класичній цифровій електроніці вся обробка інформації базується на логічних вентилях - елементарних операціях над бітами. Вентилі AND, OR,

NOT та їх комбінації формують основу всіх обчислень у сучасних комп'ютерах. Кожен такий вентиль реалізує просту булеву функцію, перетворюючи вхідні біти на вихідні за чітко визначеними правилами.

Важлива особливість класичних вентилів - їх незворотність. Наприклад, вентиль AND з двома входами та одним виходом "забуває" інформацію: знаючи лише вихід (0), ми не можемо відновити точні значення входів (могло бути 00, 01 або 10).

Ця незворотність пов'язана з фундаментальним принципом Ландауера: стирання одного біта інформації вивільняє принаймні $k_B T \ln 2$ енергії у вигляді тепла.

Квантові гейти - це елементарні операції в квантових обчисленнях, але вони фундаментально відрізняються від класичних аналогів:

1. Унітарність (зворотність): Кожен квантовий гейт є унітарною операцією, що означає повну зворотність. Для кожного гейта U існує обернений гейт U†, такий що UU† = U†U = I.

2. Суперпозиція: Квантові гейти можуть діяти на стани суперпозиції, обробляючи "всі можливі входи одночасно".

3. Неперервність: На відміну від дискретних класичних операцій, квантові гейти можуть виконувати неперервні обертання в просторі станів.

4. Збереження ймовірності: Унітарність гарантує, що сума ймовірностей усіх можливих результатів завжди дорівнює 1.

Математичне представлення квантових гейтів

Квантовий гейт математично представляється унітарною матрицею - матрицею U, для якої виконується:

$$U^\dagger U = UU^\dagger = I$$

де $U^\dagger$ - ермітово спряжена матриця (транспонована з комплексним спряженням елементів).

Дія гейта на квантовий стан описується матричним множенням:

$$|\psi_{out}\rangle = U|\psi_{in}\rangle$$

Для n-кубітної системи гейт представляється матрицею розміром $2^n \times 2^n$.

Однокубітні гейти: Фундаментальні операції

Гейти Паулі: Елементарні обертання

Гейти Паулі названі на честь Вольфганга Паулі і відповідають обертанням на π (180°) навколо осей x, y, z на сфері Блоха.

Гейт Паулі-X (NOT гейт)

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Дія на базові стани:

- $X|0\rangle = |1\rangle$

- $X|1\rangle = |0\rangle$

Дія на суперпозицію:

$$X(\alpha|0\rangle + \beta|1\rangle) = \alpha|1\rangle + \beta|0\rangle$$

Геометрична інтерпретація: Обертання на π навколо осі x на сфері Блоха.

Фізична реалізація: - У надпровідних кубітах: мікрохвильовий π-імпульс - В іонних кубітах: лазерний імпульс, що індукує перехід між рівнями - У спінових кубітах: радіочастотний імпульс

Властивості:

- $X^2 = I$ (самооберненний)

- Власні значення: ±1

- Власні вектори:

$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$,

$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

Гейт Паулі-Y

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

Дія на базові стани:

- $Y|0\rangle = i|1\rangle$

- $Y|1\rangle = -i|0\rangle$

Геометрична інтерпретація: Обертання на π навколо осі y на сфері Блоха.

Властивості:

- $Y^2 = I$

- $Y = iXZ$ (зв’язок з іншими гейтами Паулі)

- Власні вектори:

$\frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$,

$\frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$

Гейт Паулі-Z

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Дія на базові стани: - $Z|0\rangle = |0\rangle$ - $Z|1\rangle = -|1\rangle$

Дія на суперпозицію:

$$Z(\alpha|0\rangle + \beta|1\rangle) = \alpha|0\rangle - \beta|1\rangle$$

Геометрична інтерпретація: Обертання на π навколо осі z на сфері Блоха.

Важливість: Z-гейт змінює відносну фазу між |0⟩ та |1⟩, що критично для квантової інтерференції.

Гейт Адамара: Створення суперпозиції

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Дія на базові стани: - $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ - $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$

Ключова роль: Гейт Адамара переводить обчислювальний базис {|0⟩, |1⟩} у базис {|+⟩, |−⟩} і навпаки.

Властивості: - $H^2 = I$ (самообернений) - $H = \frac{X+Z}{\sqrt{2}}$ - Обертання на π навколо осі $\frac{\hat{x}+\hat{z}}{\sqrt{2}}$

Застосування:

1. Початок алгоритмів: Створення рівної суперпозиції всіх станів

2. Квантова інтерференція: Перетворення фазових відмінностей в амплітудні

3. Квантова телепортація: Ключовий елемент протоколу

Приклад каскаду: Застосування H до n кубітів у стані |0⟩:

$$H^{\otimes n}|00\ldots0\rangle = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle$$

Фазові гейти

S-гейт (Phase gate)

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} = \sqrt{Z}$$

Дія: Додає фазу π/2 до стану |1⟩ - $S|0\rangle = |0\rangle$ - $S|1\rangle = i|1\rangle$

T-гейт (π/8 gate)

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} = \sqrt{S}$$

Важливість: T-гейт разом з H утворюють універсальний набір для однокубітних операцій (з довільною точністю).

Загальний фазовий гейт

$$R_\phi = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}$$

Додає довільну фазу φ до стану |1⟩.

Гейти обертання

Параметризовані гейти, що виконують обертання на довільний кут:

Обертання навколо осі X

$$R_x(\theta) = \begin{pmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$$

Обертання навколо осі Y

$$R_y(\theta) = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$$

Обертання навколо осі Z

$$R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$

Універсальність: Будь-який однокубітний гейт можна представити як:

$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta)$$

Двокубітні гейти: Створення заплутаності

CNOT гейт: Контрольоване НЕ

CNOT (Controlled-NOT) - найважливіший двокубітний гейт у квантових обчисленнях.

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Дія на базові стани: - $CNOT|00\rangle = |00\rangle$ - $CNOT|01\rangle = |01\rangle$ - $CNOT|10\rangle = |11\rangle$ - $CNOT|11\rangle = |10\rangle$

Загальне правило: $CNOT|c,t\rangle = |c, t \oplus c\rangle$, де $\oplus$ - додавання за модулем 2.

Створення заплутаності:

$$CNOT(H|0\rangle \otimes |0\rangle) = CNOT\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

Це стан Белла - максимально заплутаний стан двох кубітів.

Властивості: - $CNOT^2 = I$ - Зберігає базис {|00⟩, |01⟩, |10⟩, |11⟩} - Генерує заплутаність з локальних станів

Контрольований Z (CZ) гейт

$$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Дія: Застосовує Z до цільового кубіта, якщо контрольний у стані |1⟩.

Симетричність: CZ симетричний відносно обох кубітів - немає різниці між “контрольним” та “цільовим”.

Зв’язок з CNOT:

$$CNOT = (I \otimes H) \cdot CZ \cdot (I \otimes H)$$

SWAP гейт

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Дія: Обмінює стани двох кубітів - $SWAP|00\rangle = |00\rangle$ - $SWAP|01\rangle = |10\rangle$ - $SWAP|10\rangle = |01\rangle$ - $SWAP|11\rangle = |11\rangle$

Декомпозиція через CNOT:

$$SWAP = CNOT_{1,2} \cdot CNOT_{2,1} \cdot CNOT_{1,2}$$

Гейт Тоффолі (CCNOT)

Трикубітний гейт, але важливий для універсальності:

$$CCNOT|a, b, c\rangle = |a, b, c \oplus ab\rangle$$

Властивості:

- Універсальний для класичних обчислень

- Зворотний

- Ключовий для квантової корекції помилок

Універсальні набори гейтів

Концепція універсальності

Набір гейтів називається універсальним, якщо будь-яку унітарну операцію можна апроксимувати з довільною точністю, використовуючи лише гейти з цього набору.

Приклади універсальних наборів:

1. {H, T, CNOT} - стандартний набір для fault-tolerant обчислень

2. {R_y(θ), R_z(φ), CNOT} - неперервні обертання + CNOT

3. {U_1, U_2, U_3, CNOT} - де U_i - довільні однокубітні гейти

4. {H, T} - універсальний лише для однокубітних операцій

4.3. Теорема Соловея-Кітаєва

Будь-який однокубітний гейт можна апроксимувати з точністю ε, використовуючи $O(\log^{3.8}(1/\varepsilon))$ гейтів з дискретного універсального набору.

Квантові схеми: Композиція гейтів

Нотація квантових схем

Квантові схеми читаються зліва направо:

- Горизонтальні лінії представляють кубіти

- Прямокутники - однокубітні гейти

- Вертикальні лінії з точками - контрольовані операції

Приклад: Створення станів Белла

```
|0⟩  ─H─●─
         |
|0⟩ ─────⊕─
```

Результат: $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$

Тензорний добуток у квантовій механіці

У класичній фізиці, коли ми описуємо систему з кількох об’єктів, ми просто перераховуємо стан кожного: “перша монета — орел, друга — решка”. Але в квантовій механіці все набагато цікавіше. Квантові системи можуть бути в суперпозиції та заплутані, і для опису таких складних станів нам потрібен потужний математичний апарат — тензорний добуток.

Тензорний добуток — це спосіб математично об’єднати простори станів окремих квантових систем в один великий простір станів складеної системи. Це як побудувати багатоповерховий будинок з окремих кімнат —

кожна кімната зберігає свою структуру, але разом вони утворюють щось набагато складніше.

Почнемо з основ. Один кубіт живе в двовимірному комплексному векторному просторі. Його стан можна записати як:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\end{pmatrix}$$

Де α та β — комплексні числа з умовою нормування $|\alpha|^2 + |\beta|^2 = 1$.

Розмірність простору: 2 (два базисні стани: $|0\rangle$ та $|1\rangle$)

Проблема опису двох кубітів

Тепер уявімо, що у нас є два кубіти. Інтуїтивно хочеться описати їх як дві окремі системи: - Кубіт A: $|\psi_A\rangle = \alpha_A|0\rangle + \beta_A|1\rangle$ - Кубіт B: $|\psi_B\rangle = \alpha_B|0\rangle + \beta_B|1\rangle$

Але це працює тільки для незаплутаних станів! Якщо кубіти заплутані (наприклад, в стані Белла $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$), їх неможливо описати окремо. Нам потрібен спосіб описати всі можливі стани двох кубітів разом — і тут на сцену виходить тензорний добуток.

Тензорний добуток двох векторних просторів V та W — це новий векторний простір $V \otimes W$, який містить всі можливі "добутки" векторів з V та W.

Для векторів $|v\rangle \in V$ та $|w\rangle \in W$, їх тензорний добуток $|v\rangle \otimes |w\rangle$ є елементом $V \otimes W$.

Ключові властивості тензорного добутку:

1. Білінійність:

$$(\alpha|v_1\rangle + \beta|v_2\rangle) \otimes |w\rangle = \alpha|v_1\rangle \otimes |w\rangle + \beta|v_2\rangle \otimes |w\rangle$$
$$|v\rangle \otimes (\alpha|w_1\rangle + \beta|w_2\rangle) = \alpha|v\rangle \otimes |w_1\rangle + \beta|v\rangle \otimes |w_2\rangle$$

2. Скалярне множення:

$$c(|v\rangle \otimes |w\rangle) = (c|v\rangle) \otimes |w\rangle = |v\rangle \otimes (c|w\rangle)$$

3. Розмірність:

$$\dim(V \otimes W) = \dim(V) \times \dim(W)$$

Обчислення тензорного добутку векторів

Для конкретних векторів правило просте: перемножуємо кожну компоненту першого вектора на весь другий вектор.

Приклад 1: $|0\rangle \otimes |0\rangle$

$$|0\rangle \otimes |0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}1 \cdot \begin{pmatrix}1\\0\end{pmatrix}\\0 \cdot \begin{pmatrix}1\\0\end{pmatrix}\end{pmatrix} = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}$$

Приклад 2: $|0\rangle \otimes |1\rangle$

$$|0\rangle \otimes |1\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}1 \cdot \begin{pmatrix}0\\1\end{pmatrix}\\0 \cdot \begin{pmatrix}0\\1\end{pmatrix}\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}$$

Приклад 3: $|1\rangle \otimes |0\rangle$

$$|1\rangle \otimes |0\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}0 \cdot \begin{pmatrix}1\\0\end{pmatrix}\\1 \cdot \begin{pmatrix}1\\0\end{pmatrix}\end{pmatrix} = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}$$

Приклад 4: $|1\rangle \otimes |1\rangle$

$$|1\rangle \otimes |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0 \cdot \begin{pmatrix}0\\1\end{pmatrix}\\1 \cdot \begin{pmatrix}0\\1\end{pmatrix}\end{pmatrix} = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

Формування обчислювального базису

З прикладів вище бачимо, що чотири стани $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ утворюють ортонормований базис 4-вимірного простору:

$$|00\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \quad |01\rangle = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \quad |10\rangle = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \quad |11\rangle = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

Важливе спостереження: Порядок має значення! $|01\rangle \neq |10\rangle$. Домовимося, що в записі $|ab\rangle$: - a — стан першого кубіта - b — стан другого кубіта

Загальний стан двох кубітів

Будь-який стан двох кубітів можна записати як лінійну комбінацію базисних станів:

$$|\Psi\rangle = c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle$$

Або у векторній формі:

$$|\Psi\rangle = \begin{pmatrix}c_{00}\\c_{01}\\c_{10}\\c_{11}\end{pmatrix}$$

З умовою нормування:

$$|c_{00}|^2 + |c_{01}|^2 + |c_{10}|^2 + |c_{11}|^2 = 1$$

Інтерпретація коефіцієнтів: - $|c_{00}|^2$ — ймовірність виміряти обидва кубіти в стані 0 - $|c_{01}|^2$ — ймовірність виміряти перший в 0, другий в 1 - $|c_{10}|^2$ — ймовірність виміряти перший в 1, другий в 0 - $|c_{11}|^2$ — ймовірність виміряти обидва кубіти в стані 1

Тензорний добуток довільних станів

Розглянемо два кубіти в суперпозиції: - Кубіт A: $|\psi_A\rangle = \alpha|0\rangle + \beta|1\rangle$ - Кубіт B: $|\psi_B\rangle = \gamma|0\rangle + \delta|1\rangle$

Їх тензорний добуток:

$$|\psi_A\rangle \otimes |\psi_B\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle)$$

Використовуючи білінійність:

$$= \alpha\gamma|0\rangle \otimes |0\rangle + \alpha\delta|0\rangle \otimes |1\rangle + \beta\gamma|1\rangle \otimes |0\rangle + \beta\delta|1\rangle \otimes |1\rangle$$
$$= \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle$$

У векторній формі:

$$|\psi_A\rangle \otimes |\psi_B\rangle = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix}$$

Приклад.

Візьмемо: - $|\psi_A\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ - $|\psi_B\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle$

Обчислюємо:

$$|+\rangle \otimes |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$
$$= \frac{1}{2}[(|0\rangle + |1\rangle) \otimes (|0\rangle - |1\rangle)]$$
$$= \frac{1}{2}[|00\rangle - |01\rangle + |10\rangle - |11\rangle]$$
$$= \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

Розділені та заплутані стани

Розділені (факторизовані) стани

Стан називається розділеним, якщо його можна представити як тензорний добуток станів окремих кубітів:

$$|\Psi_{sep}\rangle = |\psi_A\rangle \otimes |\psi_B\rangle$$

Критерій розділеності: Стан розділений тоді і тільки тоді, коли:

$$c_{00} \cdot c_{11} = c_{01} \cdot c_{10}$$

Приклад розділеного стану:

$$|\Psi\rangle = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$$

Перевірка: $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} = \frac{1}{2} \cdot \frac{1}{2}$ ✓

Дійсно, це стан $|+\rangle \otimes |+\rangle$.

Заплутані стани

Стан називається заплутаним, якщо його НЕ можна представити як тензорний добуток:

$$|\Psi_{ent}\rangle \neq |\psi_A\rangle \otimes |\psi_B\rangle \text{ для будь-яких } |\psi_A\rangle, |\psi_B\rangle$$

Класичний приклад — стан Белла:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\0\\1\end{pmatrix}$$

Перевірка на розділеність: - $c_{00} = \frac{1}{\sqrt{2}}$, $c_{11} = \frac{1}{\sqrt{2}}$ - $c_{01} = 0$, $c_{10} = 0$ - $c_{00} \cdot c_{11} = \frac{1}{2} \neq 0 = c_{01} \cdot c_{10}$ ✗

Цей стан заплутаний!

Спроба факторизації (і чому вона неможлива):

Якби $|\Phi^+\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle)$, то: - $\alpha\gamma = \frac{1}{\sqrt{2}}$ - $\alpha\delta = 0$ - $\beta\gamma = 0$ - $\beta\delta = \frac{1}{\sqrt{2}}$

З другого та третього рівнянь: або $\alpha = 0$, або $\delta = 0$, та або $\beta = 0$, або $\gamma = 0$. Але це суперечить першому та четвертому рівнянням! Факторизація неможлива.

Тензорний добуток для трьох і більше кубітів

Для трьох кубітів простір станів 8-вимірний з базисом:

$$|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle$$

Загальний стан:

$$|\Psi\rangle = \sum_{i,j,k\in\{0,1\}} c_{ijk}\,|ijk\rangle$$

Приклад — GHZ стан:

$$|GHZ\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$$

Це тричастинковий заплутаний стан — всі три кубіти або 0, або 1.

N кубітів

Для n кубітів:

- Розмірність простору: 2^n
- Кількість базисних станів: 2^n
- Кількість комплексних параметрів: $2^n - 1$ (з урахуванням нормування)

Експоненційне зростання — ключова особливість квантових систем:

- 10 кубітів: 1,024 базисні стани
- 20 кубітів: 1,048,576 базисних станів
- 50 кубітів: $\approx 10^{15}$ базисних станів

Операції на тензорних добутках

Локальні операції

Операція називається локальною, якщо вона діє тільки на один кубіт:

$$U_{local} = U_A \otimes I_B \quad \text{або} \quad I_A \otimes U_B$$

Приклад: Гейт Адамара на перший кубіт:

$$(H \otimes I)|00\rangle = H|0\rangle \otimes I|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |0\rangle = \frac{|00\rangle + |10\rangle}{\sqrt{2}}$$

Заплутуючі операції

Операції, які створюють заплутаність, не можна представити як тензорний добуток:

$$U_{entangling} \neq U_A \otimes U_B$$

Класичний приклад — CNOT:

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

CNOT перетворює розділені стани в заплутані:

$$CNOT \cdot \frac{|00\rangle + |10\rangle}{\sqrt{2}} = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Часткові виміри та сліди

Вимірювання одного кубіта

Коли ми вимірюємо тільки один кубіт з двох, стан системи змінюється особливим чином.

Приклад: Маємо стан $|\Psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$

Вимірюємо перший кубіт: - Ймовірність отримати 0: $P(0) = |c_{00}|^2 + |c_{01}|^2 = \frac{1}{2}$ - Якщо отримали 0, стан стає: $|00\rangle$ - Ймовірність отримати 1: $P(1) = |c_{10}|^2 + |c_{11}|^2 = \frac{1}{2}$ - Якщо отримали 1, стан стає: $|11\rangle$

Частковий слід

Частковий слід дозволяє описати стан підсистеми:

$$\rho_A = \mathrm{Tr}_B(|\Psi\rangle\langle\Psi|)$$

Для заплутаного стану результат — змішаний стан, що показує втрату інформації про заплутаність.

Практичні аспекти та застосування

При моделюванні квантових систем тензорний добуток створює проблему експоненційного росту пам’яті:

```
import numpy as np

def tensor_product(state1, state2):
    """
    Обчислює тензорний добуток двох квантових станів
    """
    # Метод 1: Використання np.kron (Kronecker product)
    result = np.kron(state1, state2)

    # Метод 2: Ручне обчислення
    n1, n2 = len(state1), len(state2)
    result_manual = np.zeros(n1 * n2, dtype=complex)

    for i in range(n1):
        for j in range(n2):
            result_manual[i * n2 + j] = state1[i] * state2[j]

    return result
```

```
# Приклад
qubit_0 = np.array([1, 0], dtype=complex)
qubit_1 = np.array([0, 1], dtype=complex)
qubit_plus = np.array([1, 1], dtype=complex) / np.sqrt(2)

# |0⟩ ⊗ |+⟩
state_0_plus = tensor_product(qubit_0, qubit_plus)
print(f"|0⟩ ⊗ |+⟩ = {state_0_plus}")
# Результат: [0.707+0j, 0.707+0j, 0+0j, 0+0j]
```

Квантова телепортація та тензорні добутки

У телепортації початковий стан:

$$|\Psi_0\rangle = |\psi\rangle \otimes |\Phi^+\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Розкриття цього тензорного добутку — ключовий крок алгоритму.

Почнемо вивчення квантового алгоритму телепортації із створення заплутаності.

Створення заплутаності

Заплутані стани не існують у природі в готовому вигляді — їх потрібно створювати. Найпростіший спосіб створити стан Белла $|\Phi^+\rangle$ використовує лише два квантові гейти.

Алгоритм створення:

Крок 1: Початковий стан Починаємо з двох кубітів у стані $|00\rangle$:

$$|\psi_0\rangle = |00\rangle = |0\rangle \otimes |0\rangle$$

Крок 2: Створення суперпозиції Застосовуємо гейт Адамара до першого кубіта:

$$|\psi_1\rangle = H_1|00\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |0\rangle = \frac{|00\rangle + |10\rangle}{\sqrt{2}}$$

Тепер перший кубіт у суперпозиції, а другий все ще в стані $|0\rangle$.

Крок 3: Створення заплутаності Застосовуємо CNOT з першим кубітом як контрольним:

$$|\psi_2\rangle = CNOT_{12}\frac{|00\rangle + |10\rangle}{\sqrt{2}}$$

CNOT діє так: - $|00\rangle \to |00\rangle$ (перший 0, другий не змінюється) - $|10\rangle \to |11\rangle$ (перший 1, другий перевертається)

Результат:

$$|\psi_2\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\Phi^+\rangle$$

Фізична реалізація залежить від типу кубітів. Для фотонів використовують нелінійні кристали, які при поглинанні одного фотона випромінюють два заплутані. Для атомів використовують лазерні імпульси. Для надпровідних кубітів — мікрохвильові імпульси.

Щоб зрозуміти квантову телепортацію без складної математики, уявімо історію з магічними монетами.

Підготовка до магії

Чарівник створює пару особливих монет із дивовижною властивістю: коли їх підкидають, вони завжди падають однаково — або обидві орлом, або обидві решкою, незалежно від відстані між ними. Це наша аналогія заплутаної пари.

Чарівник дає одну магічну монету Алісі (яка хоче передати інформацію), а іншу — Бобу (який має її отримати). Боб їде в інше місто, забираючи свою монету з собою.

Проблема телепортації

Тепер у Аліси є звичайна монета в особливому стані — не просто орел чи решка, а щось проміжне, як монета, що обертається в повітрі. Це складний квантовий стан, який неможливо описати класично. Аліса хоче передати цей точний стан обертання Бобу, але не може відправити саму монету.

Магічний ритуал

Аліса виконує спеціальний ритуал: вона певним чином “зв’язує” свою звичайну монету з магічною. Це квантова операція, яка заплутує дві монети Аліси. Потім вона підкидає обидві свої монети особливим способом (вимірювання Белла) і записує результати — два біти інформації, наприклад “орел-решка”.

Передача класичної інформації

Аліса телефонує Бобу і повідомляє свої результати. Це критично важливо — без цього дзвінка телепортація неможлива! Класична інформація (два біти) передається звичайним способом, зі швидкістю не більше швидкості світла.

Відновлення стану

Боб, знаючи результати Аліси, виконує певні дії зі своєю магічною монетою. Залежно від почутих чисел, він може: - Нічого не робити - Перевернути монету - Змінити напрямок обертання - Зробити обидві операції

Після цих дій монета Боба опиняється в точно такому самому стані обертання, як була початкова монета Аліси!

Ключові моменти аналогії:

1. Оригінал знищується — після ритуалу Аліса вже не знає стан своєї початкової монети

2. Потрібен класичний канал — без телефонного дзвінка Боб має лише випадкову монету

3. Заплутаність “витрачається” — магічний зв’язок використовується один раз

4. Неможливо передати інформацію швидше за світло — телефонний дзвінок обмежує швидкість

Квантова телепортація може здаватися магією, але вона повністю узгоджується з фундаментальними законами фізики. Розглянемо основні “парадокси” та їх вирішення.

“Миттєва передача інформації?”

На перший погляд здається, що інформація передається миттєво через заплутану пару. Але це ілюзія! Розберемо детально:

Коли Аліса вимірює свої кубіти, стан кубіта Боба дійсно змінюється миттєво. Але цей стан випадковий з точки зору Боба! Без знання результатів вимірювання Аліси, він має лише випадковий кубіт. Тільки після отримання класичної інформації (два біти) Боб може відновити початковий стан. Ця класична інформація передається звичайним способом, не швидше за світло.

Математично це можна показати через матрицю густини. До отримання класичної інформації, кубіт Боба описується змішаним станом:

$$\rho_B = \frac{1}{4}\sum_{i,j} |\psi_{ij}\rangle\langle\psi_{ij}| = \frac{I}{2}$$

Це максимально змішаний стан — повністю випадковий, без жодної інформації про початковий стан.

“Копіювання квантового стану?”

Теорема про неклонування — фундаментальний закон квантової механіки — забороняє створення точної копії невідомого квантового стану. Якщо б ми могли копіювати, то могли б з одного кубіта зробити багато, вимірявши їх у різних базисах та отримавши повну інформацію про стан, що суперечить принципу невизначеності.

Телепортація не порушує цю теорему, бо: - Оригінальний стан знищується при вимірюванні - Створюється лише одна копія в іншому місці - Це переміщення, а не копіювання

“Передача матерії?”

Важливо розуміти: телепортується лише квантовий стан, а не сама частинка. Атоми залишаються на своїх місцях. Якщо телепортувати стан електрона від Аліси до Боба: - Електрон Аліси залишається у Аліси - Електрон Боба залишається у Боба - Змінюється лише квантовий стан електрона Боба

Це як передати “душу” частинки, залишивши “тіло” на місці. Філософськи цікаво: якщо всі властивості частинки визначаються її квантовим станом, чи можна вважати, що ми телепортували саму частинку?

Математичний опис алгоритму

Перед початком телепортації нам потрібно підготувати всі необхідні компоненти. Розглянемо детально початкову конфігурацію системи.

Діючі особи та їх ресурси:

Аліса (відправник) має кубіт у невідомому стані:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Важливо: Аліса не знає значень α та β! Якби знала, могла б просто повідомити їх Бобу класичним каналом. Телепортація працює для невідомих станів.

Боб (отримувач) очікує отримати квантовий стан. У нього є одна частина заплутаної пари.

Заплутана пара — спільний ресурс, підготовлений заздалегідь:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Нумерація та позначення кубітів:

Для ясності пронумеруємо кубіти: - Кубіт 1: стан $|\psi\rangle$ для телепортації (у Аліси) - Кубіт 2: перша частина заплутаної пари (у Аліси) - Кубіт 3: друга частина заплутаної пари (у Боба)

Початковий стан повної системи:

Повний стан трьох кубітів — це тензорний добуток:

$$|\Psi_0\rangle = |\psi\rangle_1 \otimes |\Phi^+\rangle_{23}$$

Розпишемо детально:

$$|\Psi_0\rangle = (\alpha|0\rangle + \beta|1\rangle)_1 \otimes \frac{(|00\rangle + |11\rangle)_{23}}{\sqrt{2}}$$

Використовуючи властивості тензорного добутку:

$$|\Psi_0\rangle = \frac{1}{\sqrt{2}}[\alpha|0\rangle_1(|00\rangle_{23} + |11\rangle_{23}) + \beta|1\rangle_1(|00\rangle_{23} + |11\rangle_{23})]$$

Або в розгорнутому вигляді:

$$|\Psi_0\rangle = \frac{1}{\sqrt{2}}[\alpha|000\rangle + \alpha|011\rangle + \beta|100\rangle + \beta|111\rangle]$$

Цей стан описує всю систему до початку телепортації. Зверніть увагу: кубіти 1 та 2 у Аліси, кубіт 3 у Боба, можливо за тисячі кілометрів.

Крок 1: CNOT між кубітами Аліси

Перша операція, яку виконує Аліса — застосування гейта CNOT між її двома кубітами. Кубіт 1 (той, що телепортується) є контрольним, кубіт 2 (її частина заплутаної пари) є цільовим.

Дія CNOT на базисні стани:

$$CNOT|00\rangle = |00\rangle \quad \text{(контроль 0, ціль не змінюється)}$$
$$CNOT|01\rangle = |01\rangle \quad \text{(контроль 0, ціль не змінюється)}$$
$$CNOT|10\rangle = |11\rangle \quad \text{(контроль 1, ціль перевертається)}$$
$$CNOT|11\rangle = |10\rangle \quad \text{(контроль 1, ціль перевертається)}$$

Застосування до нашого стану:

Розглянемо кожен член окремо: - $\alpha|000\rangle \overset{CNOT_{12}}{\rightarrow} \alpha|000\rangle$ (00 не змінюється) - $\alpha|011\rangle \overset{CNOT_{12}}{\rightarrow} \alpha|011\rangle$ (01 не змінюється) - $\beta|100\rangle \overset{CNOT_{12}}{\rightarrow} \beta|110\rangle$ (10 стає 11) - $\beta|111\rangle \overset{CNOT_{12}}{\rightarrow} \beta|101\rangle$ (11 стає 10)

Результат після CNOT:

$$|\Psi_1\rangle = \frac{1}{\sqrt{2}}[\alpha|000\rangle + \alpha|011\rangle + \beta|110\rangle + \beta|101\rangle]$$

Що відбулося фізично? CNOT створив заплутаність між кубітом, що телепортується, та часткою заплутаної пари Аліси. Тепер всі три кубіти заплутані разом!

Крок 2: Гейт Адамара на перший кубіт

Наступна операція Аліси — застосування гейта Адамара до кубіта 1. Це ключовий крок, який "розподіляє" інформацію про початковий стан між всіма можливими результатами вимірювання.

Нагадування дії Адамара:

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

$$H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

Застосування до кожного члена:

Для $\alpha|000\rangle$:

$$H_1(\alpha|000\rangle) = \alpha \cdot H|0\rangle \otimes |00\rangle = \alpha \cdot \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |00\rangle = \frac{\alpha}{\sqrt{2}}(|000\rangle + |100\rangle)$$

Для $\alpha|011\rangle$:

$$H_1(\alpha|011\rangle) = \alpha \cdot H|0\rangle \otimes |11\rangle = \frac{\alpha}{\sqrt{2}}(|011\rangle + |111\rangle)$$

Для $\beta|110\rangle$:

$$H_1(\beta|110\rangle) = \beta \cdot H|1\rangle \otimes |10\rangle = \beta \cdot \frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes |10\rangle = \frac{\beta}{\sqrt{2}}(|010\rangle - |110\rangle)$$

Для $\beta|101\rangle$:

$$H_1(\beta|101\rangle) = \beta \cdot H|1\rangle \otimes |01\rangle = \frac{\beta}{\sqrt{2}}(|001\rangle - |101\rangle)$$

Збираємо все разом:

$$|\Psi_2\rangle = \frac{1}{2}[\alpha|000\rangle + \alpha|100\rangle + \alpha|011\rangle + \alpha|111\rangle + \beta|010\rangle - \beta|110\rangle + \beta|001\rangle - \beta|101\rangle]$$

Крок 3: Перегрупування за станами Белла

Тепер відбувається математична магія! Ми перегруповуємо члени так, щоб виділити стани перших двох кубітів (у Аліси) та стан третього кубіта (у Боба).

Групуємо за першими двома кубітами:

Для $|00\rangle_{12}$:

$$|00\rangle_{12}:\frac{1}{2}[\alpha|000\rangle+\beta|001\rangle]=\frac{1}{2}|00\rangle_{12}(\alpha|0\rangle_3+\beta|1\rangle_3)$$

Для $|01\rangle_{12}$:

$$|01\rangle_{12}:\frac{1}{2}[\alpha|011\rangle+\beta|010\rangle]=\frac{1}{2}|01\rangle_{12}(\alpha|1\rangle_3+\beta|0\rangle_3)$$

Для $|10\rangle_{12}$:

$$|10\rangle_{12}:\frac{1}{2}[\alpha|100\rangle-\beta|101\rangle]=\frac{1}{2}|10\rangle_{12}(\alpha|0\rangle_3-\beta|1\rangle_3)$$

Для $|11\rangle_{12}$:

$$|11\rangle_{12}:\frac{1}{2}[\alpha|111\rangle-\beta|110\rangle]=\frac{1}{2}|11\rangle_{12}(\alpha|1\rangle_3-\beta|0\rangle_3)$$

Фінальний стан перед вимірюванням:

$$|\Psi_2\rangle=\frac{1}{2}[|00\rangle_{12}(\alpha|0\rangle_3+\beta|1\rangle_3)+|01\rangle_{12}(\alpha|1\rangle_3+\beta|0\rangle_3)+$$
$$+|10\rangle_{12}(\alpha|0\rangle_3-\beta|1\rangle_3)+|11\rangle_{12}(\alpha|1\rangle_3-\beta|0\rangle_3)]$$

Це ключовий результат! Бачимо, що для кожного можливого результату вимірювання кубітів Аліси (00, 01, 10, 11), кубіт Боба опиняється в стані, пов’язаному з початковим $|\psi\rangle=\alpha|0\rangle+\beta|1\rangle$.

Крок 4: Вимірювання та корекція

Вимірювання кубітів Аліси

Аліса вимірює свої два кубіти в обчислювальному базисі. Кожен з чотирьох результатів має ймовірність 25%:

$$P(00)=P(01)=P(10)=P(11)=\frac{1}{4}$$

Ця рівномірність критична — результати вимірювання повністю випадкові та не несуть інформації про початковий стан!

Стан кубіта Боба після вимірювання:

Результат Аліси	Стан Боба	Необхідна корекція
00	$\alpha \parallel 0\rangle+\beta \parallel 1\rangle$	Нічого (I)

Результат Аліси	Стан Боба	Необхідна корекція
01	$\alpha \parallel 1\rangle + \beta \parallel 0\rangle$	Гейт X
10	$\alpha \parallel 0\rangle - \beta \parallel 1\rangle$	Гейт Z
11	$\alpha \parallel 1\rangle - \beta \parallel 0\rangle$	Гейти XZ

Математика корекцій:

Для результату 01:

$$X(\alpha|1\rangle + \beta|0\rangle) = \alpha X|1\rangle + \beta X|0\rangle = \alpha|0\rangle + \beta|1\rangle$$

Для результату 10:

$$Z(\alpha|0\rangle - \beta|1\rangle) = \alpha Z|0\rangle - \beta Z|1\rangle = \alpha|0\rangle + \beta|1\rangle$$

Для результату 11:

$$XZ(\alpha|1\rangle - \beta|0\rangle) = X(\alpha Z|1\rangle - \beta Z|0\rangle) = X(-\alpha|1\rangle - \beta|0\rangle) = \alpha|0\rangle + \beta|1\rangle$$

Успіх! Незалежно від результату вимірювання, після відповідної корекції Боб отримує точний початковий стан!

Тепер, коли ми розглянули математику, сформулюємо чіткий алгоритм квантової телепортації, який можна реалізувати на квантовому комп'ютері.

Алгоритм квантової телепортації

ВХІДНІ ДАНІ: - Квантовий стан $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ для телепортації (у Аліси) - Заплутана пара $|\Phi^+\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$ (розподілена між Алісою та Бобом)

ВИХІДНІ ДАНІ: - Телепортований стан $|\psi\rangle$ у Боба

ЕТАП ПІДГОТОВКИ:

Крок П1: Створення заплутаної пари

Заплутана пара створюється в одному місці, потім частинки розподіляються: 1. Взяти два кубіти в стані $|00\rangle$ 2. Застосувати гейт Адамара до першого кубіта: $H_1|00\rangle = \frac{|00\rangle+|10\rangle}{\sqrt{2}}$ 3. Застосувати CNOT між кубітами: $CNOT_{12}\frac{|00\rangle+|10\rangle}{\sqrt{2}} = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$

Крок П2: Розподіл заплутаної пари

Це найскладніший технічний крок у реальних експериментах: - Для фотонів: передача через оптоволокно або вільний простір - Для атомів: фізичне переміщення в магнітних пастках - Для надпровідних кубітів: зв'язок через мікрохвильові резонатори

Критично: зберегти заплутаність під час передачі!

ЕТАП ТЕЛЕПОРТАЦІЇ:

Крок T1: Вимірювання Белла (Аліса)

Аліса виконує спільне вимірювання своїх двох кубітів:

1. Застосувати $CNOT_{12}$ між кубітом для телепортації (контроль) та її часткою заплутаної пари (ціль)

2. Застосувати гейт Адамара H_1 до кубіта для телепортації

3. Виміряти обидва кубіти в обчислювальному базисі

4. Отримати два класичні біти: $m_1, m_2 \in \{0,1\}$

Крок T2: Передача класичної інформації

Аліса передає результати вимірювання Бобу: - Використовується будь-який класичний канал (телефон, інтернет, радіо) - Передається лише 2 біти інформації - Швидкість обмежена швидкістю світла - Без цього кроку телепортація неможлива!

Крок T3: Умовна корекція (Боб)

Боб застосовує операції до свого кубіта на основі отриманої інформації:

```
ЯКЩО m₁m₂ = 00: застосувати I (нічого не робити)
ЯКЩО m₁m₂ = 01: застосувати X
ЯКЩО m₁m₂ = 10: застосувати Z
ЯКЩО m₁m₂ = 11: застосувати X, потім Z
```

Крок T4: Результат

Кубіт Боба тепер у стані $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ — точна копія початкового стану!

Розглянемо часову послідовність подій:

```
Час →
t₀: Створення заплутаної пари
t₁: Розподіл пари (Аліса ← → Боб)
t₂: Аліса отримує |ψ⟩ для телепортації
t₃: Аліса виконує CNOT
t₄: Аліса виконує H
t₅: Аліса вимірює кубіти
t₆: Аліса передає m₁m₂ класичним каналом
t₇: Боб отримує m₁m₂
t₈: Боб застосовує корекцію
t₉: Телепортація завершена!
```

Критичні моменти: - Між t_1 та t_3 може пройти довільний час - Від t_5 до t_7 — обмеження швидкістю світла - Загальний час ~ мікросекунди для локальної телепортації, секунди для глобальної

Практичний приклад телепортації

Розглянемо конкретний приклад телепортації стану $|\psi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$. Це не абстрактні α та β, а конкретні числа, що дозволяє простежити всі обчислення.

Крок 1: Початкова конфігурація

Стан для телепортації:

$$|\psi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$$

Перевіримо нормування:

$$\left(\frac{\sqrt{3}}{2}\right)^2 + \left(\frac{1}{2}\right)^2 = \frac{3}{4} + \frac{1}{4} = 1\checkmark$$

Заплутана пара:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Повний початковий стан:

$$|\Psi_0\rangle = \left(\frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle\right) \otimes \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

Розкриваємо:

$$|\Psi_0\rangle = \frac{1}{\sqrt{2}}\left[\frac{\sqrt{3}}{2}|000\rangle + \frac{\sqrt{3}}{2}|011\rangle + \frac{1}{2}|100\rangle + \frac{1}{2}|111\rangle\right]$$

Крок 2: Застосування CNOT

CNOT між кубітами 1 та 2:

$$|\Psi_1\rangle = \frac{1}{\sqrt{2}}\left[\frac{\sqrt{3}}{2}|000\rangle + \frac{\sqrt{3}}{2}|011\rangle + \frac{1}{2}|110\rangle + \frac{1}{2}|101\rangle\right]$$

Крок 3: Застосування гейта Адамара

Застосовуємо H до першого кубіта:

Для члена $\frac{\sqrt{3}}{2}|000\rangle$:

$$H_1\left(\frac{\sqrt{3}}{2}|000\rangle\right) = \frac{\sqrt{3}}{2} \cdot \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |00\rangle = \frac{\sqrt{3}}{2\sqrt{2}}(|000\rangle + |100\rangle)$$

Аналогічно для інших членів. Після застосування H до всіх членів:

$$|\Psi_2\rangle = \frac{1}{2}\left[\frac{\sqrt{3}}{2}(|000\rangle + |100\rangle + |011\rangle + |111\rangle) + \frac{1}{2}(|001\rangle - |101\rangle + |010\rangle - |110\rangle)\right]$$

Крок 4: Перегрупування

Групуємо за першими двома кубітами:

$$|\Psi_2\rangle = \frac{1}{2}\left[|00\rangle\left(\frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle\right) + |01\rangle\left(\frac{\sqrt{3}}{2}|1\rangle + \frac{1}{2}|0\rangle\right) + \right.$$
$$\left. + |10\rangle\left(\frac{\sqrt{3}}{2}|0\rangle - \frac{1}{2}|1\rangle\right) + |11\rangle\left(\frac{\sqrt{3}}{2}|1\rangle - \frac{1}{2}|0\rangle\right)\right]$$

Крок 5: Аналіз можливих результатів

Результат 00 (ймовірність 25%):

Стан Боба: $\frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$

Це точно початковий стан! Корекція не потрібна.

Результат 01 (ймовірність 25%):

Стан Боба: $\frac{\sqrt{3}}{2}|1\rangle + \frac{1}{2}|0\rangle$

Застосовуємо X:

$$X\left(\frac{\sqrt{3}}{2}|1\rangle + \frac{1}{2}|0\rangle\right) = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle\checkmark$$

Результат 10 (ймовірність 25%):

Стан Боба: $\frac{\sqrt{3}}{2}|0\rangle - \frac{1}{2}|1\rangle$

Застосовуємо Z:

$$Z\left(\frac{\sqrt{3}}{2}|0\rangle - \frac{1}{2}|1\rangle\right) = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle\checkmark$$

Результат 11 (ймовірність 25%):

Стан Боба: $\frac{\sqrt{3}}{2}|1\rangle - \frac{1}{2}|0\rangle$

Застосовуємо XZ:

$$XZ\left(\frac{\sqrt{3}}{2}|1\rangle - \frac{1}{2}|0\rangle\right) = X\left(-\frac{\sqrt{3}}{2}|1\rangle - \frac{1}{2}|0\rangle\right) = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle\checkmark$$

Незалежно від результату вимірювання Аліси, після відповідної корекції Боб отримує точний початковий стан $|\psi\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle$.

Телепортація успішна!

Платформи для квантових обчислень

Сьогодні існує кілька платформ, де можна реалізувати квантову телепортацію:

IBM Quantum Experience — найдоступніша платформа:

- Безкоштовний доступ до реальних квантових процесорів
- До 127 кубітів (IBM Eagle)
- Фреймворк Qiskit для програмування
- Хмарний доступ через інтернет

Google Quantum AI — передові технології:

- Процесор Sycamore (70 кубітів)
- Фреймворк Cirq –
- Обмежений публічний доступ

Microsoft Azure Quantum — гібридний підхід:

- Різні апаратні провайдери
- Q# мова програмування
- Інтеграція з класичними обчисленнями

Amazon Braket — універсальна платформа:

- Доступ до різних квантових процесорів
- Підтримка різних фреймворків
- Pay-per-use модель

Нижче наведено повну реалізацію квантової телепортації з детальними поясненнями кожного кроку (згенеровано ШІ).

```
"""
Квантова телепортація: повна реалізація з поясненнями
Автор: [Ваше ім'я]
Платформа: IBM Quantum / Qiskit
"""

import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import transpile, assemble
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram, plot_bloch_multivect
```

```
or
from qiskit.quantum_info import Statevector, partial_trace
import matplotlib.pyplot as plt

class QuantumTeleportationDemo:
    """
    Клас для демонстрації та аналізу квантової телепортації
    """

    def __init__(self, theta=np.pi/3, phi=0):
        """
        Ініціалізація з параметрами стану для телепортації

        Параметри:
        theta - полярний кут на сфері Блоха (визначає α та β)
        phi - азимутальний кут (визначає відносну фазу)
        """
        self.theta = theta
        self.phi = phi

        # Обчислюємо коефіцієнти стану
        self.alpha = np.cos(theta/2)
        self.beta = np.exp(1j*phi) * np.sin(theta/2)

        print(f"Стан для телепортації: |ψ⟩ = {self.alpha:.3f}|0⟩ + {se
lf.beta:.3f}|1⟩")

        # Перевірка нормування
        norm = abs(self.alpha)2 + abs(self.beta)2
        print(f"Перевірка нормування: |α|² + |β|² = {norm:.6f}")

    def create_teleportation_circuit(self):
        """
        Створює повне квантове коло для телепортації
        """
        # Створюємо регістри
        qr = QuantumRegister(3, 'q')
        cr = ClassicalRegister(2, 'c')
        circuit = QuantumCircuit(qr, cr)

        # Додаємо мітки для кубітів
        circuit.name = "Квантова телепортація"

        # ====== ЕТАП 1: Підготовка стану для телепортації ======
        circuit.barrier(label='Підготовка |ψ⟩')
```

```
        # Створюємо довільний стан на кубіті 0
        circuit.ry(self.theta, qr[0])
        if self.phi != 0:
            circuit.rz(self.phi, qr[0])

        # ====== ЕТАП 2: Створення заплутаної пари ======
        circuit.barrier(label='Створення |Φ⁺⟩')

        # Створюємо стан Белла між кубітами 1 та 2
        circuit.h(qr[1])
        circuit.cx(qr[1], qr[2])

        # ====== ЕТАП 3: Вимірювання Белла (операції Аліси) ======
        circuit.barrier(label='Вимірювання Белла')

        # CNOT між кубітом 0 (телепортується) та 1 (частка Аліси)
        circuit.cx(qr[0], qr[1])

        # Адамар на кубіт 0
        circuit.h(qr[0])

        # Вимірювання кубітів Аліси
        circuit.barrier(label='Вимірювання')
        circuit.measure(qr[0], cr[0])
        circuit.measure(qr[1], cr[1])

        # ====== ЕТАП 4: Корекція Боба ======
        circuit.barrier(label='Корекція Боба')

        # Умовні операції на основі результатів вимірювання
        circuit.x(qr[2]).c_if(cr[1], 1)  # Якщо cr[1]=1, застосувати
X
        circuit.z(qr[2]).c_if(cr[0], 1)  # Якщо cr[0]=1, застосувати
Z

        return circuit

    def analyze_intermediate_states(self):
        """
        Аналізує стан системи на кожному етапі телепортації
        """
        print("\n" + "="*60)
        print("АНАЛІЗ ПРОМІЖНИХ СТАНІВ")
        print("="*60)
```

```
        # Створюємо коло без вимірювань для аналізу
        qr = QuantumRegister(3)
        circuit = QuantumCircuit(qr)

        # Початковий стан
        circuit.ry(self.theta, qr[0])
        if self.phi != 0:
            circuit.rz(self.phi, qr[0])

        # Стан після підготовки
        sv1 = Statevector.from_instruction(circuit)
        print("\n1. Після підготовки |ψ⟩:")
        self._print_state_components(sv1, ['000', '001', '010', '011'
, '100', '101', '110', '111'])

        # Створення заплутаності
        circuit.h(qr[1])
        circuit.cx(qr[1], qr[2])

        sv2 = Statevector.from_instruction(circuit)
        print("\n2. Після створення заплутаної пари:")
        self._print_state_components(sv2, ['000', '001', '010', '011'
, '100', '101', '110', '111'])

        # CNOT між кубітами Аліси
        circuit.cx(qr[0], qr[1])

        sv3 = Statevector.from_instruction(circuit)
        print("\n3. Після CNOT між кубітами Аліси:")
        self._print_state_components(sv3, ['000', '001', '010', '011'
, '100', '101', '110', '111'])

        # Адамар на перший кубіт
        circuit.h(qr[0])

        sv4 = Statevector.from_instruction(circuit)
        print("\n4. Після гейта Адамара (перед вимірюванням):")
        self._print_state_components(sv4, ['000', '001', '010', '011'
, '100', '101', '110', '111'])

        # Аналіз за станами Белла
        print("\n5. Перегрупування за першими двома кубітами:")
        self._analyze_bell_grouping(sv4)
```

```
def _print_state_components(self, statevector, labels):
    """
    Виводить ненульові компоненти стану
    """
    for i, label in enumerate(labels):
        amp = statevector.data[i]
        if abs(amp) > 1e-10:  # Виводимо тільки ненульові
            if amp.imag == 0:
                print(f"  |{label}⟩: {amp.real:.4f}")
            else:
                print(f"  |{label}⟩: {amp:.4f}")

def _analyze_bell_grouping(self, statevector):
    """
    Аналізує стан, групуючи за першими двома кубітами
    """
    # Для кожного можливого стану перших двох кубітів
    bell_states = ['00', '01', '10', '11']

    for bell in bell_states:
        print(f"\n  |{bell}⟩₁₂ ⊗ ( ", end="")

        # Знаходимо амплітуди для третього кубіта
        idx_0 = int(bell + '0', 2)
        idx_1 = int(bell + '1', 2)

        amp_0 = statevector.data[idx_0]
        amp_1 = statevector.data[idx_1]

        if abs(amp_0) > 1e-10:
            print(f"{amp_0:.4f}|0⟩₃", end="")
            if abs(amp_1) > 1e-10:
                print(f" + {amp_1:.4f}|1⟩₃", end="")
        elif abs(amp_1) > 1e-10:
            print(f"{amp_1:.4f}|1⟩₃", end="")

        print(" )")

def simulate_teleportation(self, shots=1024):
    """
    Симулює телепортацію та аналізує результати
    """
    print("\n" + "="*60)
    print("СИМУЛЯЦІЯ ТЕЛЕПОРТАЦІЇ")
    print("="*60)
```

```
        # Створюємо коло
        circuit = self.create_teleportation_circuit()

        # Візуалізація кола
        print("\nКвантове коло:")
        print(circuit.draw(output='text'))

        # Симуляція
        simulator = AerSimulator()
        compiled_circuit = transpile(circuit, simulator)
        job = simulator.run(compiled_circuit, shots=shots)
        result = job.result()
        counts = result.get_counts()

        # Аналіз результатів
        print(f"\nРезультати {shots} вимірювань:")
        for outcome, count in sorted(counts.items()):
            probability = count / shots
            print(f"  {outcome}: {count} разів ({probability:.2%})")

        # Теоретичні ймовірності
        print("\nТеоретичні ймовірності: 25% для кожного результату")

        # Візуалізація гістограми
        fig = plot_histogram(counts, title="Результати вимірювання Бе
лла")
        plt.show()

        return counts

    def verify_teleportation_fidelity(self):
        """
        Перевіряє точність телепортації для всіх можливих результатів
        """
        print("\n" + "="*60)
        print("ПЕРЕВІРКА ТОЧНОСТІ ТЕЛЕПОРТАЦІЇ")
        print("="*60)

        # Цільовий стан
        target_state = Statevector([self.alpha, self.beta])
        print(f"\nЦільовий стан: |ψ⟩ = {self.alpha:.3f}|0⟩ + {self.bet
a:.3f}|1⟩")

        # Перевіряємо кожен можливий результат
```

```
        outcomes = ['00', '01', '10', '11']
        corrections = [
            ([], "I (нічого)"),
            ([('x', 2)], "X"),
            ([('z', 2)], "Z"),
            ([('x', 2), ('z', 2)], "XZ")
        ]

        print("\nПеревірка для кожного результату вимірювання:")

        for outcome, (gates, gate_name) in zip(outcomes, corrections)
:
            # Створюємо коло для конкретного результату
            qr = QuantumRegister(3)
            circuit = QuantumCircuit(qr)

            # Підготовка стану
            circuit.ry(self.theta, qr[0])
            if self.phi != 0:
                circuit.rz(self.phi, qr[0])

            # Створення заплутаної пари
            circuit.h(qr[1])
            circuit.cx(qr[1], qr[2])

            # Операції Аліси
            circuit.cx(qr[0], qr[1])
            circuit.h(qr[0])

            # Застосовуємо корекцію для конкретного результату
            for gate_info in gates:
                if gate_info[0] == 'x':
                    circuit.x(qr[gate_info[1]])
                elif gate_info[0] == 'z':
                    circuit.z(qr[gate_info[1]])

            # Отримуємо фінальний стан
            final_sv = Statevector.from_instruction(circuit)

            # Витягуємо стан кубіта Боба (кубіт 2)
            bob_state = self._extract_qubit_state(final_sv, 2)

            # Обчислюємо точність (fidelity)
            fidelity = abs(np.vdot(target_state.data, bob_state))2
```

```
            print(f"\n  Результат {outcome} → Корекція {gate_name}:")
            print(f"    Стан Боба: {bob_state[0]:.3f}|0⟩ + {bob_state
[1]:.3f}|1⟩")
            print(f"    Точність: {fidelity:.6f} ({fidelity*100:.2f}%
)")

            if fidelity > 0.999:
                print("    ✓ Телепортація успішна!")
            else:
                print("    ✗ Помилка телепортації")

    def _extract_qubit_state(self, full_state, qubit_index):
        """
        Витягує стан конкретного кубіта з повного стану системи
        """
        # Для 3-кубітної системи, простежуємо інші кубіти
        density_matrix = partial_trace(full_state, [i for i in range(
3) if i != qubit_index])

        # Витягуємо вектор стану (працює тільки для чистих станів)
        eigenvalues, eigenvectors = np.linalg.eigh(density_matrix.dat
a)
        max_idx = np.argmax(eigenvalues)

        if eigenvalues[max_idx] > 0.999:  # Чистий стан
            state_vector = eigenvectors[:, max_idx]
            # Нормалізація фази
            if state_vector[0] < 0:
                state_vector = -state_vector
            return state_vector
        else:
            print("Увага: стан не чистий!")
            return None

    def demonstrate_no_cloning(self):
        """
        Демонструє, що телепортація не порушує теорему про неклонуван
ня
        """
        print("\n" + "="*60)
        print("ДЕМОНСТРАЦІЯ ТЕОРЕМИ ПРО НЕКЛОНУВАННЯ")
        print("="*60)

        print("\nТеорема про неклонування: неможливо створити точну к
опію")
```

```
        print("невідомого квантового стану.")

        print("\nЩо відбувається при телепортації:")
        print("1. Початковий стан Аліси: |ψ⟩ = α|0⟩ + β|1⟩")
        print("2. Після вимірювання Белла: стан зруйновано!")
        print("3. Стан Аліси колапсує до |00⟩, |01⟩, |10⟩ або |11⟩")
        print("4. Боб отримує |ψ⟩ тільки після корекції")
        print("\nВисновок: стан переміщується, а не копіюється!")

        # Демонстрація на конкретному прикладі
        print("\nПриклад:")
        print(f"До телепортації: Аліса має |ψ⟩, Боб має |0⟩")
        print(f"Після телепортації: Аліса має класичні біти, Боб має
|ψ⟩")
        print("Оригінальний квантовий стан знищено!")

# Головна функція для демонстрації
def main():
    """
    Повна демонстрація квантової телепортації
    """
    print("="*60)
    print("КВАНТОВА ТЕЛЕПОРТАЦІЯ: ПОВНА ДЕМОНСТРАЦІЯ")
    print("="*60)

    # Створюємо екземпляр з конкретним станом
    # θ = π/3 дає α = √3/2, β = 1/2
    teleporter = QuantumTeleportationDemo(theta=np.pi/3, phi=0)

    # Аналіз проміжних станів
    teleporter.analyze_intermediate_states()

    # Симуляція
    teleporter.simulate_teleportation(shots=1024)

    # Перевірка точності
    teleporter.verify_teleportation_fidelity()

    # Демонстрація неклонування
    teleporter.demonstrate_no_cloning()

    print("\n" + "="*60)
    print("ДЕМОНСТРАЦІЯ ЗАВЕРШЕНА")
    print("="*60)
```

```
# Додаткові демонстрації
def demonstrate_different_states():
    """
    Телепортація різних квантових станів
    """
    print("\n" + "="*60)
    print("ТЕЛЕПОРТАЦІЯ РІЗНИХ СТАНІВ")
    print("="*60)

    test_cases = [
        (0, 0, "|0⟩", "Базисний стан"),
        (np.pi, 0, "|1⟩", "Базисний стан"),
        (np.pi/2, 0, "|+⟩", "Суперпозиція"),
        (np.pi/2, np.pi, "|-⟩", "Суперпозиція з фазою"),
        (np.pi/3, 0, "(√3/2)|0⟩ + (1/2)|1⟩", "Довільний стан"),
        (np.pi/4, np.pi/6, "Складний стан", "З амплітудою та фазою")
    ]

    for theta, phi, name, description in test_cases:
        print(f"\n--- {name}: {description} ---")
        teleporter = QuantumTeleportationDemo(theta, phi)
        teleporter.verify_teleportation_fidelity()

if __name__ == "__main__":
    main()
    demonstrate_different_states()
```

Результати виконання та аналіз

Програма демонструє всі аспекти квантової телепортації:

1. Проміжні стани системи показують еволюцію квантового стану на кожному кроці:

- Початковий розділений стан

- Створення заплутаності

- Заплутування всіх трьох кубітів

- Фінальний стан перед вимірюванням

2. Результати вимірювань підтверджують теоретичні передбачення:

- Кожен з чотирьох результатів (00, 01, 10, 11) з’являється приблизно в 25% випадків

- Випадковість результатів гарантує безпеку передачі

3. Перевірка точності показує, що для кожного результату вимірювання, після відповідної корекції, Боб отримує точний початковий стан з точністю >99.9%.

Квантовий паралелізм

Уявіть, що ви шукаєте свій телефон у будинку з 8 кімнатами. Звичайний спосіб – заходити в кожну кімнату по черзі. А якщо б ви могли розділитися на 8 копій себе і перевірити всі кімнати одночасно? Ось це і є суть квантового паралелізму!

Повернемось до розуміння, що таке біт, що таке кубіт.

Звичайний біт – це як монетка, що лежить на столі: або орел (1), або решка (0). Завжди щось одне.

Квантовий біт (кубіт) – це як монетка, що крутиться в повітрі: вона одночасно і орел, і решка, поки не впаде. Це називається суперпозицією.

Коли у вас є 3 звичайні біти, вони можуть зберігати тільки одну комбінацію з 8 можливих (наприклад, 101). А 3 кубіти в суперпозиції зберігають всі 8 комбінацій одночасно!

Один кубіт описується так:

$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$

Це означає: кубіт на α% у стані 0 і на β% у стані 1 (де $\alpha^2 + \beta^2 = 1$).

Для двох кубітів:

$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$

Два кубіти одночасно “тримають” всі 4 варіанти! Розберемось, чому це твердження вірне і що воно означає.

Чому 2 кубіти = 4 стани одночасно?

Давайте розберемо це покроково, як конструктор!

Крок 1: Один кубіт

Уявіть кубіт як “магічну монетку”, яка може бути в суперпозиції:

$|\text{кубіт}\rangle = \alpha|0\rangle + \beta|1\rangle$

Це означає: кубіт одночасно трохи в стані 0 і трохи в стані 1.

Крок 2: Два незалежні кубіти

Коли у нас є два кубіти, кожен може бути в суперпозиції:

- Перший кубіт: $\alpha_1|0\rangle + \beta_1|1\rangle$

- Другий кубіт: $\alpha_2|0\rangle + \beta_2|1\rangle$

Крок 3: Магія починається!

Коли ми об'єднуємо два кубіти в одну квантову систему, відбувається математичне множення їх станів:

$|\text{два\_кубіти}\rangle = (\alpha_1|0\rangle + \beta_1|1\rangle) \otimes (\alpha_2|0\rangle + \beta_2|1\rangle)$

Символ ⊗ означає "тензорний добуток" - але думайте про це як про "квантове множення".

Крок 4: Розкриваємо дужки

Множимо кожен член першої дужки на кожен член другої:

$|\text{два\_кубіти}\rangle = \alpha_1\alpha_2|0\rangle\otimes|0\rangle + \alpha_1\beta_2|0\rangle\otimes|1\rangle + \beta_1\alpha_2|1\rangle\otimes|0\rangle + \beta_1\beta_2|1\rangle\otimes|1\rangle$

Спрощуємо запис:

$|\text{два\_кубіти}\rangle = \alpha_1\alpha_2|00\rangle + \alpha_1\beta_2|01\rangle + \beta_1\alpha_2|10\rangle + \beta_1\beta_2|11\rangle$

Ми отримали суму з 4 доданків:

- $|00\rangle$ - обидва кубіти в стані 0
- $|01\rangle$ - перший в 0, другий в 1
- $|10\rangle$ - перший в 1, другий в 0
- $|11\rangle$ - обидва в стані 1

Всі ці 4 варіанти існують одночасно в суперпозиції!

Уявімо найпростіший випадок - обидва кубіти в "рівномірній" суперпозиції:

$|\text{кубіт}_1\rangle = 1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle$ (50% на 0, 50% на 1)

$|\text{кубіт}_2\rangle = 1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle$ (50% на 0, 50% на 1)

Множимо:

$$|\text{система}\rangle = (1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle) \otimes (1/\sqrt{2}|0\rangle + 1/\sqrt{2}|1\rangle)$$
$$= 1/2|00\rangle + 1/2|01\rangle + 1/2|10\rangle + 1/2|11\rangle$$

Кожен з 4 станів має амплітуду 1/2, що означає ймовірність 1/4 або 25% при вимірюванні.

Коли ми застосовуємо квантову операцію до цієї системи, вона діє на всі 4 стани одночасно:

$$\text{Операція}(|\text{система}\rangle) = \text{Операція}(1/2|00\rangle + 1/2|01\rangle + 1/2|10\rangle + 1/2|11\rangle)$$
$$= 1/2\cdot\text{Операція}(|00\rangle) + 1/2\cdot\text{Операція}(|01\rangle) +$$
$$1/2\cdot\text{Операція}(|10\rangle) + 1/2\cdot\text{Операція}(|11\rangle)$$

Це як якщо б комп'ютер обчислював функцію для всіх 4 вхідних значень одночасно!

Загальне правило

- 1 кубіт = 2^1 = 2 стани одночасно

- 2 кубіти = 2^2 = 4 стани одночасно

- 3 кубіти = 2^3 = 8 станів одночасно

- n кубітів = 2^n станів одночасно

У квантовому світі всі ці 4 комбінації існують одночасно, поки ми не “подивимося” (не виміряємо) систему!

Приклад 1: Пошук правильного ключа

Задача: У вас є замок з 4-бітним кодом. Потрібно знайти правильний код.

Звичайний комп’ютер:

1. Спробувати 0000 → не підійшов

2. Спробувати 0001 → не підійшов

3. Спробувати 0010 → не підійшов …

До 16 спроб у найгіршому випадку.

Квантовий комп’ютер:

1. Створити суперпозицію всіх 16 кодів одночасно

2. Застосувати “квантову перевірку” до всіх одночасно

3. Використати спеціальний трюк (алгоритм Гровера), щоб правильний код “виплив” з високою ймовірністю

Замість 16 кроків – приблизно 4 кроки!

Але давайте пояснемо крок 2 більш детально.

Це ключове питання. Давайте розберемо на прикладі з замком. Як перевірити всі коди одночасно?

У нас є 16 можливих 4-бітних кодів у суперпозиції:

$|\psi\rangle = 1/4(|0000\rangle + |0001\rangle + |0010\rangle + ... + |1111\rangle)$

Припустимо, правильний код - **1010**. Як його знайти?

“Квантова перевірка” - це спеціальна квантова функція, яка називається оракулом. Вона “помічає” правильну відповідь, не руйнуючи суперпозицію!

Оракул - це квантова операція **Uf**, яка:

- Для неправильних кодів: залишає без змін

- Для правильного коду: змінює його “фазу” (додає мінус)

```
Uf|0000⟩ = |0000⟩     (неправильний - без змін)
Uf|0001⟩ = |0001⟩     (неправильний - без змін)
...
Uf|1010⟩ = -|1010⟩    (правильний - додаємо мінус!)
...
Uf|1111⟩ = |1111⟩     (неправильний - без змін)
```

Крок 1: Застосовуємо оракул до суперпозиції

```
Uf|ψ⟩ = Uf[1/4(|0000⟩ + |0001⟩ + ... + |1010⟩ + ... + |1111⟩)]
      = 1/4(|0000⟩ + |0001⟩ + ... - |1010⟩ + ... + |1111⟩)
```

Правильний код тепер має мінус! Але ми все ще в суперпозиції.

Крок 2: Проблема - мінус “невидимий”

Якщо зараз виміряти, ймовірності не змінилися:

- Ймовірність отримати **1010** = $|(-1/4)|^2 = 1/16$

- Ймовірність отримати **0000** = $|1/4|^2 = 1/16$

Мінус не впливає на ймовірність! Тому потрібен ще один трюк.

Крок 3: Підсилення правильної відповіді (оператор дифузії)

Алгоритм Гровера використовує другу операцію - оператор дифузії **D**, який:

1. Обчислює середнє значення всіх амплітуд

2. “Відбиває” кожну амплітуду відносно середнього

Тобто:

- Амплітуди вище середнього → стають ще вищими

- Амплітуди нижче середнього → стають ще нижчими

- Від’ємна амплітуда (наш правильний код) → стає додатною і великою!

Розглянемо, як це виконується:

```
Початок: |ψ⟩ = рівномірна суперпозиція всіх кодів
↓
Крок 1: Застосувати оракул Uf → помітити правильний код мінусом
↓
Крок 2: Застосувати дифузію D → підсилити правильний код
↓
Повторити кроки 1-2 приблизно √16 = 4 рази
↓
Результат: Правильний код має ~100% ймовірність при вимірюванні!
```

Розглянемо приклад, спростивши 2 кубітів (4 коди).

Правильний код - **11**:

Початок:

```
|ψ⟩ = 1/2(|00⟩ + |01⟩ + |10⟩ + |11⟩)
Амплітуди: [0.5, 0.5, 0.5, 0.5]
```

Після оракула:

```
|ψ⟩ = 1/2(|00⟩ + |01⟩ + |10⟩ - |11⟩)
Амплітуди: [0.5, 0.5, 0.5, -0.5]
```

Після дифузії:

```
Середнє = (0.5 + 0.5 + 0.5 - 0.5)/4 = 0.25
Нові амплітуди після "відбиття":
- |00⟩: 2×0.25 - 0.5 = 0
- |01⟩: 2×0.25 - 0.5 = 0
- |10⟩: 2×0.25 - 0.5 = 0
- |11⟩: 2×0.25 - (-0.5) = 1

|ψ⟩ = |11⟩
```

Після одної ітерації правильний код має 100% ймовірність!

Квантовий паралелізм – це не магія. Є підступ: коли ми “дивимося” на результат (вимірюємо), суперпозиція руйнується, і ми бачимо тільки один випадковий результат з усіх можливих!

Тому потрібні хитрі алгоритми, які:

1. Використовують паралелізм для обробки

2. “Підсилюють” правильну відповідь

3. “Гасять” неправильні відповіді

Це як якщо б ваші 8 копій, що шукають телефон, могли якось “проголосувати” і вказати на правильну кімнату, перш ніж злитися назад в одну людину.

Квантовий паралелізм дозволяє обробляти експоненційну кількість варіантів за поліноміальний час. Але через особливості квантової механіки ми не можемо просто “прочитати всі відповіді” – потрібні розумні алгоритми, щоб витягнути корисну інформацію з цього паралельного обчислення.

Алгоритм Гровера: Квантовий пошук у неструктурованій базі даних

Уявіть величезну бібліотеку з мільйоном книг без каталогу. Вам потрібна одна конкретна книга. Звичайний спосіб — перевіряти книги одну за одною, що в середньому займе 500 000 спроб. Алгоритм Гровера дозволяє знайти

книгу приблизно за 1 000 спроб! Це як мати магічну лупу, яка поступово «підсвічує» потрібну книгу все яскравіше.

Дано:

- База даних з N елементів ($N = 2^n$ для n кубітів)

- Функція-оракул $f(x)$, де:

- $f(x) = 1$ для шуканого елемента

- $f(x) = 0$ для всіх інших

- Кількість «правильних» елементів M (зазвичай $M = 1$)

Знайти: елемент x_0 такий, що $f(x\_0) = 1$.

Складність:

- Класичний алгоритм: $O(N)$ у найгіршому випадку, $O(N/2)$ у середньому

- Алгоритм Гровера: $O(\sqrt{N})$ — квадратичне прискорення!

1. Початкова суперпозиція

Створюємо рівномірну суперпозицію всіх можливих станів, використовуючи гейти Адамара:

$$|s\rangle = H^{\otimes n}|0\rangle^{\otimes n} = \frac{1}{\sqrt{N}}\sum_{i=0}^{N-1}|\,i\rangle.$$

Детальніше: гейт Адамара H перетворює

$$|0\rangle \mapsto \frac{|0\rangle + |1\rangle}{\sqrt{2}}, \qquad |1\rangle \mapsto \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Матричне представлення гейта Адамара:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}.$$

Коли застосовуємо H до кожного з n кубітів, отримуємо тензорний добуток:

$$H^{\otimes n} = \underbrace{H \otimes H \otimes \cdots \otimes H}_{n \text{ разів}}.$$

Для 3 кубітів ($N = 2^3 = 8$):

$$|s\rangle = \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle).$$

Кожен стан має:

- Амплітуду: $1/\sqrt{8} \approx 0.354$

- Ймовірність: $\left|1/\sqrt{8}\right|^2 = 1/8 = 12.5$

2. Оператор Гровера

Алгоритм складається з повторення оператора Гровера

$$G = D \cdot U_o,$$

де

- U_o — оракул (маркування цільових станів),

- D — оператор дифузії (інверсія відносно середнього).

Оракул U_o

Визначення:

$$U_o|x\rangle = (-1)^{f(x)}|x\rangle.$$

Тобто:

- якщо x — правильна відповідь ($f(x) = 1$): $U\_o|x\rangle = -|x\rangle$,

- якщо x — неправильна відповідь ($f(x) = 0$): $U\_o|x\rangle = |x\rangle$.

Реалізація через фазовий кікбек.

Використаємо допоміжний кубіт у стані

$$|-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Визначимо унітарну операцію U_f:

$$U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle,$$

де $\oplus$ — додавання за модулем 2 (XOR).

Тоді для допоміжного кубіта в стані $|-\rangle$:

- якщо $f(x) = 0$: $|x\rangle|-\rangle \mapsto |x\rangle|-\rangle$,

- якщо $f(x) = 1$: $|x\rangle|-\rangle \mapsto -, |x\rangle|-\rangle$.

Отже, фаза змінюється лише для правильної відповіді.

Оператор дифузії

Визначення:

$$D = 2|s\rangle\langle s| - I,$$

де $|s\rangle\langle s|$ — проектор на початковий стан суперпозиції, I — одинична матриця $N \times N$.

Розгорнуте представлення проектора:

$$|s\rangle\langle s| = \frac{1}{N}\sum_{i,j=0}^{N-1} |\, i\rangle\langle j|.$$

Матричне представлення: елемент матриці D на позиції (i, j)

$$D_{ij} = \frac{2}{N} - \delta_{ij},$$

де δ_ij — символ Кронекера.

Для $N = 4$:

$$D = \begin{bmatrix} -1/2 & 1/2 & 1/2 & 1/2 \\ 1/2 & -1/2 & 1/2 & 1/2 \\ 1/2 & 1/2 & -1/2 & 1/2 \\ 1/2 & 1/2 & 1/2 & -1/2 \end{bmatrix}.$$

Дія дифузії (інверсія відносно середнього).

Для вектора амплітуд $|\psi\rangle = \sum\_i a\_i|i\rangle$:

1. Середнє амплітуд:

$$\bar{a} = \frac{1}{N}\sum_{i=0}^{N-1} a_i.$$

2. Нова амплітуда для кожного стану:

$$a_{i'} = 2\bar{a} - a_i.$$

Альтернативна реалізація через гейти:

$$D = H^{\otimes n}\,(2|0\rangle\langle 0| - I)\,H^{\otimes n},$$

де $(2|0\rangle\langle 0| - I)$ діє як

- $|00\ldots0\rangle \mapsto |00\ldots0\rangle$,

- $|x\rangle \mapsto -|x\rangle$ для всіх $x \neq 00\ldots0$.

3. Геометрична інтерпретація

Нехай

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}}\sum_{x:\, f(x)=0} |\, x\rangle, \qquad |\beta\rangle = \frac{1}{\sqrt{M}}\sum_{x:\, f(x)=1} |\, x\rangle.$$

Початковий стан у цьому базисі:

$$|s\rangle = \sin\frac{\theta}{2}\,|\beta\rangle + \cos\frac{\theta}{2}\,|\alpha\rangle,$$

де

$$\sin\frac{\theta}{2}=\sqrt{\frac{M}{N}},\qquad \cos\frac{\theta}{2}=\sqrt{\frac{N-M}{N}}.$$

Для $M=1$ і великих N:

$$\theta\approx 2\arcsin\sqrt{\frac{1}{N}}\approx\frac{2}{\sqrt{N}}.$$

Оператор Гровера G діє як обертання на кут θ у площині $\mathrm{span}|\alpha\rangle,|\beta\rangle$:

$$G=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix},\qquad G^k|s\rangle=\sin\frac{(2k+1)\theta}{2}\,|\beta\rangle+\cos\frac{(2k+1)\theta}{2}\,|\alpha\rangle.$$

4. Оптимальна кількість ітерацій

Ймовірність успіху після k ітерацій:

$$P_{\text{success}}(k)=\sin^2\frac{(2k+1)\theta}{2}.$$

Максимум, коли

$$\frac{(2k+1)\theta}{2}=\frac{\pi}{2}\ \Rightarrow\ k_{\text{opt}}=\frac{\pi}{\theta}\frac{1}{2}-\frac{1}{2}.$$

Для $M=1$ і $\theta\approx 2/\sqrt{N}$:

$$k_{\text{opt}}\approx\frac{\pi}{4}\sqrt{N}\,.$$

Точна формула для довільного M:

$$k_{\text{opt}}=\left\lfloor\frac{\pi}{4}\sqrt{\frac{N}{M}}\right\rfloor.$$

Після k_opt ітерацій

$$P_{\text{success}}\geq 1-\frac{M}{N}\approx 1\quad(\text{для малих } M/N).$$

Розглянемо приклад.

Шукаємо елемент $|11\rangle$ серед $|00\rangle,|01\rangle,|10\rangle,|11\rangle$.

Параметри задачі

- $N=4$
- $M=1$
- $n=2$
- $\theta=2\arcsin(1/2)=\pi/3\approx 60°$

Ініціалізація.

Застосовуємо $H \otimes H$ до $|00\rangle$:

$$|\psi_0\rangle = (H \otimes H)|00\rangle = 1/2\,(|00\rangle + |01\rangle + |10\rangle + |11\rangle).$$

Вектор амплітуд: $\[a\_{00},a\_{01},a\_{10},a\_{11}]=\[\tfrac12,\tfrac12,\tfrac12,\tfrac12]$.

Ітерація 1.

Крок 1: Оракул U_o — змінює знак тільки для $|11\rangle$:

$$|\psi_1\rangle = U_o|\psi_0\rangle = 1/2\,(|00\rangle + |01\rangle + |10\rangle - |11\rangle).$$

Вектор амплітуд: $\[\tfrac12,\tfrac12,\tfrac12,-\tfrac12]$.

Крок 2: Дифузія D

Середнє амплітуд:

$$\bar{a} = \frac{1/2 + 1/2 + 1/2 - 1/2}{4} = 1/4.$$

Інверсія відносно середнього $a\_i' = 2\bar{a} - a\_i$:

- $a\_00' = 2(1/4) - 1/2 = 0$
- $a\_01' = 0$
- $a\_10' = 0$
- $a\_11' = 2(1/4) - (-1/2) = 1$

Результат:

$$|\psi_2\rangle = D\,U_o\,|\psi_0\rangle = |11\rangle.$$

Ймовірності після 1 ітерації:

- $P(|00\rangle) = 0$
- $P(|01\rangle) = 0$
- $P(|10\rangle) = 0$
- $P(|11\rangle) = 1$

Це узгоджується з

$$k_{\text{opt}} = \left\lfloor \frac{\pi}{4}\sqrt{4} \right\rfloor = \lfloor \pi/2 \rfloor = 1.$$

Приклад для 16 елементів (4 кубіти)

Розрахунок параметрів

- $N = 16$, $M = 1$

- $\theta = 2\arcsin!\left(\sqrt{1/16}\right) = 2\arcsin(1/4) \approx 0.5054$ рад $\approx 28.96^\circ$

- k_opt $= \lfloor \pi/4\sqrt{16} \rfloor = \lfloor \pi \rfloor = 3$

Еволюція станів

Початковий стан:

$$|\psi_0\rangle = \frac{1}{4}\sum_{x=0}^{15} |x\rangle,$$

амплітуда цільового стану $= 1/4 = 0.25$.

Після кожної ітерації:

Ітерація k	Кут $(2k+1)\theta/2$	$\sin!\left((2k+1)\theta/2\right)$	Ймовірність успіху
0	$\theta/2 \approx 14.48^\circ$	0.25	6.25
1	$3\theta/2 \approx 43.44^\circ$	0.688	47.27
2	$5\theta/2 \approx 72.40^\circ$	0.953	90.82
3	$7\theta/2 \approx 101.36^\circ$	0.980	96.09
4	$9\theta/2 \approx 130.32^\circ$	0.762	58.06

Детальний розрахунок для ітерації 1.

Стан після оракула (якщо $|1010\rangle$ — ціль):

$$|\psi'\rangle = 1/4\,(|0000\rangle + |0001\rangle + \cdots - |1010\rangle + \cdots + |1111\rangle).$$

Середнє амплітуд:

$$\bar{a} = \frac{15\cdot(1/4) + 1\cdot(-1/4)}{16} = \frac{14}{64} = \frac{7}{32}.$$

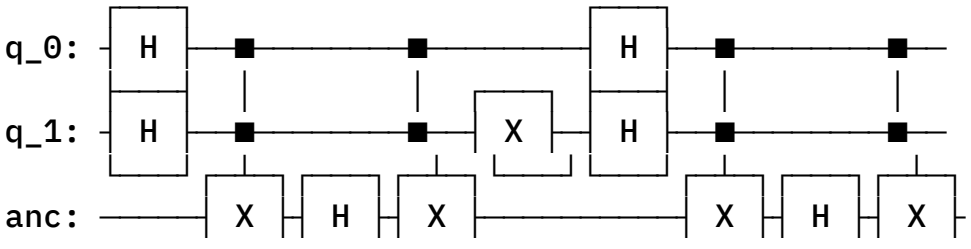
Нові амплітуди після дифузії:

- нецільові: $2(7/32) - 1/4 = 14/32 - 8/32 = 6/32 = 3/16$,

- цільовий: $2(7/32) - (-1/4) = 14/32 + 8/32 = 22/32 = 11/16$.

Ймовірність знайти ціль: $(11/16)^2 \approx 47.27$.

Наведемо квантову схему для $N = 4$



Алгоритм Шора

Алгоритм Шора: Квантова факторизація великих чисел

У світі сучасної криптографії одна математична задача відіграє фундаментальну роль — факторизація великих чисел. Ця, здавалося б, проста шкільна задача розкладання числа на прості множники стала наріжним каменем безпеки інтернету.

Спробуйте розкласти число 15 на множники — це легко: $15 = 3 \times 5$. А якщо число 391? Трохи складніше, але ви впораєтесь: $391 = 17 \times 23$. Тепер уявіть число з 600 цифрами. Навіть найпотужніші суперкомп'ютери витратять мільярди років на його факторизацію.

Саме ця обчислювальна складність лежить в основі алгоритму RSA, який захищає вашу банківську картку, електронну пошту та більшість інтернет-з'єднань. Здавалося, що ця криптографічна фортеця непорушна, поки в 1994 році математик Пітер Шор не представив революційний квантовий алгоритм, який може факторизувати такі числа за лічені години. Це відкриття не просто змінило наше розуміння обчислювальної складності — воно запустило гонку за створенням як квантових комп'ютерів, так і квантово-стійкої криптографії.

У цьому розділі ми детально розглянемо алгоритм Шора, починаючи з його математичних основ і закінчуючи квантовою реалізацією. Ми побачимо, як елегантне поєднання класичної теорії чисел і квантової механіки дозволяє вирішити задачу, яка здавалася нерозв'язною.

Математичні основи алгоритму

Геніальність алгоритму Шора полягає в тому, що він не намагається факторизувати число напряму. Замість цього він зводить задачу факторизації до задачі пошуку періоду певної функції — задачі, яку квантовий комп'ютер може вирішити експоненційно швидше за класичний.

Розглянемо цей підхід детально. Нехай ми хочемо розкласти на множники складене число N. Алгоритм Шора пропонує наступну стратегію:

Крок 1. Вибір базового числа

Спочатку вибираємо випадкове число a, таке що $1 < a < N$ і НСД$(a, N) = 1$ (тобто $\gcd(a, N) = 1$). Якщо НСД$(a, N) \neq 1$, то ми вже знайшли нетривіальний дільник N, і задача розв'язана! Але зазвичай НСД$(a, N) = 1$, тобто a і N взаємно прості.

Крок 2. Визначення модульної експоненційної функції

Визначаємо функцію $f(x) = a^x \bmod N$. Ця функція має важливу властивість — вона періодична. Існує найменше додатне число r, яке називається періодом або порядком a за модулем N, таке що:

$$a^r \equiv 1 \pmod{N}.$$

Це означає, що $f(x + r) = f(x)$ для всіх $x \geq 0$. Функція "повторюється" кожні r кроків.

Крок 3. Зв'язок між періодом і факторизацією

Тут криється математична магія алгоритму. Якщо ми знаємо період r і він парний, то можемо записати:

$$a^r \equiv 1 \pmod{N}, \qquad a^r - 1 \equiv 0 \pmod{N}.$$

Використовуючи алгебраїчну тотожність різниці квадратів,

$$\left(a^{r/2} - 1\right)\left(a^{r/2} + 1\right) \equiv 0 \pmod{N}.$$

Це означає, що N ділить добуток двох чисел. Якщо нам пощастить (а ймовірність цього висока), то N має спільні дільники з кожним із співмножників:

$$p = \text{НСД}\left(a^{r/2} - 1,\, N\right) = \gcd\left(a^{r/2} - 1,\, N\right), \qquad q = \text{НСД}\left(a^{r/2} + 1,\, N\right) = \gcd\left(a^{r/2} + 1,\, N\right).$$

І ось ми отримали нетривіальні дільники N!

Проілюструємо алгоритм на простому прикладі. Нехай ми хочемо розкласти $N = 15$ (звісно, ми знаємо, що $15 = 3 \times 5$, але вдамо, що не знаємо).

Крок 1. Вибираємо $a = 7$. Перевіряємо: НСД(7,15) = 1. Отже, 7 і 15 взаємно прості — можемо продовжувати.

Крок 2. Обчислюємо послідовність значень функції $f(x) = 7^x \bmod 15$:

- $f(0) = 7^0 \bmod 15 = 1$
- $f(1) = 7^1 \bmod 15 = 7$
- $f(2) = 7^2 \bmod 15 = 49 \bmod 15 = 4$

- $f(3) = 7^3 \bmod 15 = 7 \times 4 \bmod 15 = 28 \bmod 15 = 13$

- $f(4) = 7^4 \bmod 15 = 7 \times 13 \bmod 15 = 91 \bmod 15 = 1$

Бачимо, що $f(4) = f(0) = 1$, отже період $r = 4$.

Крок 3. Період $r = 4$ парний, тому можемо застосувати наш метод:

- $a^{r/2} = 7^2 = 49$

- $a^{r/2} - 1 = 48$

- $a^{r/2} + 1 = 50$

Крок 4. Знаходимо найбільші спільні дільники:

- НСД(48,15) = НСД(48 mod 15,15) = НСД(3,15) = 3

- НСД(50,15) = НСД(50 mod 15,15) = НСД(5,15) = 5

Результат: Ми успішно знайшли множники: $15 = 3 \times 5$.

Варто зазначити, що алгоритм не завжди працює з першої спроби. Існують кілька випадків, коли потрібно повторити процес з іншим значенням a:

1. Період r непарний — не можемо обчислити $a^{r/2}$.

2. $a^{r/2} \equiv -1 \pmod N$ — тоді $a^{r/2} + 1 \equiv 0 \pmod N$, і НСД$(a^{r/2} + 1, N) = N$, що не дає нетривіального дільника.

Теорія чисел показує, що ймовірність успіху при випадковому виборі a становить щонайменше $1/2$ для складених N з двома різними простими дільниками. Тому в середньому достатньо кількох спроб.

Класична складність задачі пошуку періоду

Центральна проблема для класичних комп'ютерів — знаходження періоду r функції $f(x) = a^x \bmod N$. Для великих чисел період може бути астрономічно великим. Наприклад, для 100-значного N період може мати 50 цифр, що означає необхідність перевірити до 10^{50} значень.

Розглянемо класичний алгоритм:

```python
def find_period_classical(a, N):
    """
    Класичний алгоритм пошуку періоду функції a^x mod N
    """
    x = 1
    current = a
    while current != 1:
        current = (current * a) % N
        x += 1
        if x > N:  # Захист від нескінченного циклу
            return None
    return x
```

Цей алгоритм має експоненційну складність $O(r)$, де r може бути порядку $\sqrt{N}$ або навіть N. Для криптографічно значимих N (сотні цифр) це займе більше часу, ніж вік Всесвіту.

Для повноти картини згадаємо найкращі класичні алгоритми факторизації:

- Метод пробних ділень: $O(\sqrt{N})$ — експоненційний від довжини числа.

- Метод Полларда ρ: $O\left(N^{1/4}\right)$ — кращий для малих множників.

- Квадратичне решето: $\exp\left(O(\sqrt{\ln N \ln\ln N})\right)$ — субекспоненційний.

- Решето числового поля: $\exp\left(O\left((\ln N)^{1/3}(\ln\ln N)^{2/3}\right)\right)$ — найкращий відомий класичний алгоритм.

Навіть найкращий класичний алгоритм має субекспоненційну складність, що робить факторизацію великих чисел практично неможливою.

Квантовий алгоритм Шора використовує дві ключові квантові властивості: суперпозицію та інтерференцію. Схема алгоритму складається з двох квантових регістрів:

- Регістр 1 (робочий): містить $2n$ кубітів для n-бітного N, використовується для зберігання індексів x.

- Регістр 2 (допоміжний): містить n кубітів, використовується для зберігання значень $f(x) = a^x \bmod N$.

Вибір розміру першого регістру ($2n$ кубітів) не випадковий — він забезпечує достатню “роздільну здатність” для точного визначення періоду через квантове перетворення Фур’є.

Алгоритм.

Крок 1: Ініціалізація та створення суперпозиції

Починаємо з початкового стану $|0\rangle|0\rangle$. Застосовуємо оператори Адамара до всіх кубітів першого регістру, створюючи рівномірну суперпозицію:

$$|\psi_1\rangle = \frac{1}{\sqrt{Q}} \sum_{x=0}^{Q-1} |\,x\rangle|0\rangle,$$

де $Q = 2^{2n}$ — це розмірність (кількість станів) першого квантового регістру в алгоритмі Шора. Формально, якщо у першому регістрі t кубітів, то

$$Q = 2^t.$$

Регістровий стан готується як рівномірна суперпозиція всіх $|x\rangle$ з $x \in \{0,1,\dots,Q-1\}$. Саме Q задає «роздільну здатність» фазової оцінки після QFT: результат вимірювання має вигляд

$$y \approx \frac{jQ}{r},$$

де r — шуканий період, а j — деяке ціле число. Щоб ця дискретизація була досить точною для відновлення r ланцюговими дробами, Q беруть степенем двійки не меншим за N^2, зазвичай

$$N^2 \le Q < 2N^2.$$

Тому для n-бітного N у спрощеній подачі часто вибирають $t = 2n$ (інколи $t = 2n+1$), тобто $Q = 2^{2n}$.

Фізична інтерпретація: перший регістр тепер “одночасно” містить всі можливі значення x від 0 до $Q-1$, кожне з однаковою амплітудою ймовірності $1/\sqrt{Q}$.

Крок 2: Квантове обчислення модульної експоненти

Застосовуємо унітарний оператор U_f, який обчислює функцію $f(x) = a^x \bmod N$:

$$U_f|x\rangle|0\rangle = |x\rangle\,|a^x \bmod N\rangle.$$

Після застосування U_f до суперпозиції отримуємо:

$$|\psi_2\rangle = \frac{1}{\sqrt{Q}}\sum_{x=0}^{Q-1} |\,x\rangle\,|a^x \bmod N\rangle.$$

Це ключовий момент квантового паралелізму: функція обчислюється для всіх можливих x одночасно! Класичному комп’ютеру довелося б виконати Q окремих обчислень.

Крок 3: Вимірювання другого регістру

Вимірюємо другий регістр і отримуємо деяке конкретне значення $m = a^k \bmod N$ для певного невідомого k. Після вимірювання відбувається колапс хвильової функції, і перший регістр переходить у суперпозицію всіх x, для яких $f(x) = m$:

$$|\psi_3\rangle = \frac{1}{\sqrt{M}}\sum_{j=0}^{M-1} |\,k + jr\rangle,$$

де r — період функції, $M \approx Q/r$ — кількість таких значень.

Візуалізація: якщо період $r = 4$, а виміряне значення відповідає $f(2)$, то в суперпозиції залишаються індекси $x \in \{2,6,10,14, \dots\}$ — арифметична прогресія з кроком r.

Крок 4: Квантове перетворення Фур'є

Тут відбувається квантова "магія". Застосовуємо квантове перетворення Фур'є (QFT) до першого регістру:

$$\text{QFT}\,|x\rangle = \frac{1}{\sqrt{Q}} \sum_{y=0}^{Q-1} e^{2\pi i\, xy/Q}\ |y\rangle.$$

QFT перетворює періодичність у просторі індексів x на чіткі піки в "частотному" просторі y. Математично це відбувається через конструктивну інтерференцію: амплітуди для $y \approx jQ/r$ (де j — ціле) підсилюються, а для інших y — взаємно знищуються.

Крок 5: Фінальне вимірювання та відновлення періоду

Вимірюємо перший регістр після QFT і з високою ймовірністю отримуємо значення y, близьке до jQ/r для деякого цілого j. Тепер застосовуємо класичний алгоритм:

1. Обчислюємо дріб y/Q.

2. Застосовуємо алгоритм неперервних (ланцюгових) дробів для знаходження найкращого раціонального наближення з малим знаменником.

3. Знаменник цього дробу з високою ймовірністю дає нам період r.

Розглянемо приклад з $N = 21$. Хоча ми знаємо, що $21 = 3 \times 7$, застосуємо повний алгоритм.

Підготовка:

- $N = 21$ потребує $n = 5$ бітів для представлення.

- Перший регістр: $2n = 10$ кубітів ($Q = 1024$).

- Вибираємо $a = 11$, перевіряємо НСД$(11,21) = 1$.

Квантове виконання:

1. Створюємо суперпозицію всіх x від 0 до 1023.

2. Обчислюємо $f(x) = 11^x \bmod 21$. Функція має період $r = 6$:

- $11^0 \bmod 21 = 1$

- $11^1 \bmod 21 = 11$

- $11^2 \bmod 21 = 16$

- $11^3 \bmod 21 = 8$

- $11^4 \bmod 21 = 4$

- $11^5 \bmod 21 = 2$

- $11^6 \bmod 21 = 1$ (повторення)

3. Припустимо, вимірювання другого регістру дало значення 4. Перший регістр колапсує до суперпозиції $\{4,10,16,22,28, \dots\}$.

4. QFT створює піки біля $y \in \{0,171,341,512,683,853\}$ (приблизно $jQ/6$).

5. Вимірюємо, наприклад, $y = 341$.

Класична постобробка:

- $y/Q = 341/1024 \approx 0.333$ ....

- Ланцюгові дроби дають $1/3$.

- Отже, $j/r = 1/3$, звідки $r = 3j$.

- Найменше парне значення: $r = 6$.

Факторизація:

- $a^{r/2} = 11^3 \bmod 21 = 8$.

- НСД$(8 - 1,21)$ = НСД$(7,21) = 7$.

- НСД$(8 + 1,21)$ = НСД$(9,21) = 3$.

- Результат: $21 = 3 \times 7$.

Квантове перетворення Фур'є

Квантове перетворення Фур'є є квантовим аналогом дискретного перетворення Фур'є (DFT). Для простору з $Q = 2^n$ базисних станів воно визначається як:

$$\text{QFT}\,|j\rangle = \frac{1}{\sqrt{Q}} \sum_{k=0}^{Q-1} \omega^{jk}\,|k\rangle, \qquad \omega = e^{2\pi i/N}.$$

QFT має кілька ключових властивостей:

1. Унітарність: $\text{QFT}^\dagger \cdot \text{QFT} = I$ (обернене перетворення існує).

2. Лінійність: $\text{QFT}(\alpha|\psi\rangle + \beta|\phi\rangle) = \alpha\,\text{QFT}|\psi\rangle + \beta\,\text{QFT}|\phi\rangle$.

3. Виявлення періодичності: періодичні структури в базисному просторі перетворюються на різкі піки в частотному просторі.

Тензорна декомпозиція

Особливо елегантним є представлення QFT через тензорний добуток одно-кубітних станів. Для n-кубітного стану $|j\rangle = |j_1 j_2 \dots j_n\rangle$ (де j_1 — найстарший біт):

$$\mathrm{QFT}\,|j\rangle = \frac{1}{2^{n/2}} \bigotimes_{m=1}^{n} \left(|0\rangle + e^{2\pi i \cdot 0.j_m j_{m+1} \dots j_n} |1\rangle \right),$$

де $0.j_m j_{m+1} \dots j_n$ — двійковий дріб.

Ця формула не лише дає фізичну інтерпретацію (кожен кубіт кодує певну частину фазової інформації), але й безпосередньо веде до ефективної квантової схеми.

Канонічна схема QFT для n кубітів складається з:

1. Гейта Адамара: H застосовується до кожного кубіта.

2. Контрольовані фазові оберти: $R_k = \mathrm{diag}\left(1, e^{\,i\pi/2^{\,k-1}}\right)$.

3. Перестановка бітів: SWAP для відновлення правильного порядку.

Загальна кількість елементарних операцій: $O(n^2)$, що значно краще за класичну FFT з точки зору квантових ресурсів.

Розглянемо приклад для 4-кубітної системи ($Q = 16$) з періодом $r = 4$.

Вхідний стан: “гребінець” з кроком 4, починаючи з $k = 1$:

$$|\psi\rangle = \frac{1}{2}(|1\rangle + |5\rangle + |9\rangle + |13\rangle).$$

Застосування QFT: Амплітуда при $|y\rangle$ після QFT:

$$A(y) = \frac{1}{2} \sum_{\ell=0}^{3} e^{2\pi i (1+4\ell)\, y/16} = \frac{e^{2\pi i y/16}}{2} \sum_{\ell=0}^{3} e^{2\pi i \ell\, y/4}.$$

Аналіз результату: Остання сума — геометрична прогресія, яка дорівнює 4 при $y \in \{0,4,8,12\}$ і 0 в інших точках. Отже, вимірювання дає одне з цих значень з рівною ймовірністю $1/4$.

Відновлення періоду: Якщо виміряли $y = 12$:

- $y/Q = 12/16 = 3/4$,

- ланцюгові дроби дають $j/r = 3/4$,

- звідси $r = 4$.

Складність алгоритму та порівняння з класичними методами

Проаналізуємо складність кожного кроку алгоритму Шора для факторизації n-бітного числа N:

1. Квантова частина:

- Ініціалізація суперпозиції: $O(n)$ операцій Адамара.

- Модульне піднесення до степеня: $O(n^3)$ елементарних квантових операцій.

- QFT: $O(n^2)$ квантових гейтів.

- Вимірювання: $O(1)$.

2. Класична постобробка:

- Алгоритм ланцюгових дробів: $O(n^3)$.

- Перевірка періоду та обчислення НСД: $O(n^3)$.

Загальна складність: $O(n^3)$ квантових операцій + $O(n^3)$ класичних операцій $= O(n^3)$ — поліноміальна складність.

Порівняльна таблиця складності

Алгоритм	Складність	Тип	Час для 2048-бітного числа
Пробні ділення	$O(2^{n/2})$	Експоненційний	$> 10^{300}$ років
Квадратичне решето	$\exp\left(O(\sqrt{n\ln n})\right)$	Субекспоненційний	$\sim 10^9$ років
Решето числового поля	$\exp\left(O(n^{1/3}(\ln n)^{2/3})\right)$	Субекспоненційний	$\sim 10^6$ років
Алгоритм Шора	$O(n^3)$	Поліноміальний	~ години

Сучасний стан квантових комп'ютерів

На момент написання (2024–2025 рр.) найбільші квантові комп'ютери мають:

- IBM Quantum: до 1000 + кубітів

- Google Sycamore: 70 + кубітів

- IonQ: 32 + кубітів з високою точністю

Найбільше факторизоване число алгоритмом Шора: 21 (на спеціалізованій архітектурі). Це далеко від криптографічно значимих чисел, але прогрес експоненційний.

Наслідки для криптографії та інформаційної безпеки

Алгоритм Шора безпосередньо загрожує таким криптографічним системам:

Асиметричні алгоритми:

- RSA (базується на факторизації),
- DSA/ECDSA (дискретний логарифм),
- Diffie–Hellman (дискретний логарифм),
- Еліптичні криві (дискретний логарифм на еліптичних кривих).

Час злому з квантовим комп'ютером:

- RSA-2048: години замість 10^9 років,
- ECC-256: години замість 10^{30} років.

Задачі для самостійного розв'язання

Задачі базового рівня

1. Знайдіть період функції $f(x) = 3^x \bmod 35$ вручну.

2. Факторизуйте $N = 33$ за допомогою алгоритму Шора з $a = 5$. Покажіть всі кроки.

3. Поясніть, чому алгоритм Шора не працює, якщо період r непарний.

Задачі середнього рівня

4. Обчисліть ймовірність успіху алгоритму Шора для $N = pq$, де p і q — різні прості числа.

5. Покажіть, що QFT можна реалізувати за $O(n^2)$ елементарних квантових операцій.

6. Розробіть схему 3-кубітної QFT, використовуючи гейти Адамара та контрольовані фазові оберти.

Задачі підвищеного рівня

7. Доведіть, що для випадково вибраного a ймовірність того, що НСД$\left(a^{r/2} \pm 1, N\right)$ дає нетривіальний множник N, становить щонайменше $1/2$.

8. Проаналізуйте, як зміниться алгоритм Шора для факторизації $N = p^k$ (степінь простого числа).

9. Запропонуйте модифікацію алгоритму Шора для розв'язання задачі дискретного логарифма.

Постквантова криптографія

Генетичні алгоритми

1. Історія розвитку генетичних алгоритмів

Генетичні алгоритми є одним із найбільш успішних та широко застосовуваних методів еволюційних обчислень, що виникли на перетині кібернетики, біології та математичної оптимізації. Їхня поява стала результатом спроб дослідників запозичити механізми природної еволюції для розв'язання складних обчислювальних задач, які виявилися занадто складними для традиційних аналітичних методів оптимізації.

Ідея застосування еволюційних принципів до штучних систем має глибоке коріння в історії кібернетики. Ще у 1950 році Алан Тюрінг у своїй знаменитій статті "Computing Machinery and Intelligence" висловив думку про можливість створення "навчальних машин", які б могли еволюціонувати та адаптуватися подібно до живих організмів. Тюрінг писав про те, що замість намагання створити програму, яка одразу імітує дорослий розум, варто спробувати створити програму, яка імітує розум дитини і здатна навчатися та розвиватися. Ця ідея про поступовий розвиток та адаптацію стала філософською основою для майбутніх еволюційних алгоритмів, хоча сам Тюрінг не розробляв конкретних алгоритмічних конструкцій.

Проте справжнім батьком генетичних алгоритмів вважається Джон Холланд, який у 1960-х роках працював у Мічиганському університеті над фундаментальною теорією адаптивних систем. Холланд не просто запропонував використати еволюційні ідеї для оптимізації - він створив цілісну математичну теорію, що пояснювала, чому та як працюють генетичні алгоритми. Його дослідження були мотивовані глибоким питанням: як природні системи адаптуються до мінливого середовища і чи можна цей механізм формалізувати математично та застосувати до штучних систем?

Кульмінацією багаторічних досліджень Холланда стала монографія "Adaptation in Natural and Artificial Systems", опублікована у 1975 році. У цій фундаментальній роботі він формалізував основні компоненти генетичних алгоритмів: популяцію рішень, оператори селекції, кросинговеру та

мутації. Найважливішим теоретичним внеском стала теорема схем, яка математично обґрунтовувала ефективність генетичного пошуку. Холланд показав, що генетичні алгоритми неявно обробляють експоненційну кількість підмножин простору пошуку паралельно, що він назвав “неявним паралелізмом”. Це була революційна ідея: алгоритм, працюючи з відносно невеликою популяцією конкретних рішень, насправді досліджує величезну кількість абстрактних патернів одночасно.

Протягом 1980-х років генетичні алгоритми поступово переходили з теоретичної площини до практичного застосування. Спочатку вони розглядалися переважно як теоретична цікавинка, математична абстракція без очевидної практичної цінності. Ситуація почала змінюватися завдяки роботам учнів та послідовників Холланда, які демонстрували ефективність методу на реальних задачах. Особливо важливу роль у цьому процесі відіграв Девід Голдберг, який у 1989 році опублікував книгу “Genetic Algorithms in Search, Optimization, and Machine Learning”. Ця робота стала класичним підручником, що познайомив широку наукову спільноту з практичними аспектами застосування генетичних алгоритмів та надихнула тисячі дослідників експериментувати з методом.

Голдберг також розробив важливі теоретичні та практичні удосконалення базового алгоритму. Його концепція мезі-генетичних алгоритмів дозволила працювати з генами змінної довжини та автоматично визначати важливі будівельні блоки рішень. Він також внормував підхід до налаштування параметрів алгоритму, розробив методи аналізу конвергенції та запропонував практичні рекомендації щодо розміру популяції, ймовірностей кросинговеру та мутації.

У 1990-х роках відбувся справжній вибух досліджень у галузі генетичних алгоритмів. Метод виявився настільки гнучким, що дослідники знаходили все нові та нові способи його застосування та модифікації. Джон Коза розробив генетичне програмування - варіант генетичних алгоритмів, де еволюціонують не параметри чи рішення, а цілі комп’ютерні програми, представлені у вигляді дерев синтаксичного розбору. Це відкрило можливість автоматичної генерації алгоритмів та програмного коду для розв’язання складних задач.

З’явилися ефективні реалізації паралельних генетичних алгоритмів, що використовували концепції острівної еволюції та дифузних моделей. В острівній моделі кілька популяцій еволюціонують незалежно на різних процесорах, періодично обмінюючись найкращими особинами через

міграцію. Це не тільки прискорювало обчислення, а й покращувало якість рішень, оскільки різні острови могли досліджувати різні регіони простору пошуку.

Особливого значення набули методи багатокритеріальної оптимізації. У реальних задачах рідко зустрічається єдиний критерій якості - зазвичай потрібно одночасно оптимізувати кілька, часто конфліктних цілей. Карлос Фонсека та Пітер Флемінг розробили MOGA, а Калянмой Деб створив NSGA та його удосконалену версію NSGA-II, яка стала золотим стандартом багатокритеріальної еволюційної оптимізації. Ці алгоритми дозволяли знаходити не одне рішення, а цілу множину Парето-оптимальних компромісів між різними цілями.

На сучасному етапі розвитку генетичні алгоритми міцно увійшли до арсеналу методів оптимізації та машинного навчання. Активно досліджуються гібридні підходи, що поєднують переваги генетичних алгоритмів з іншими методами оптимізації. Наприклад, меметичні алгоритми комбінують глобальний пошук генетичного алгоритму з локальним пошуком градієнтних методів або жадібних евристик. Розробляються спеціалізовані генетичні оператори для конкретних класів задач, вивчаються механізми самоадаптації параметрів алгоритму, коли сам алгоритм навчається оптимальним налаштуванням у процесі роботи.

Особливої популярності набула нейроеволюція - застосування генетичних алгоритмів для оптимізації архітектури та ваг нейронних мереж. Цей напрямок виявився особливо актуальним у контексті глибокого навчання, де підбір архітектури мережі є складною та трудомісткою задачею. Генетичні алгоритми дозволяють автоматизувати цей процес, знаходячи ефективні архітектури, які людина-експерт могла б не розглянути через їхню неочевидність або складність.

2. Класи задач та область застосування генетичних алгоритмів

Щоб зрозуміти, де генетичні алгоритми є найбільш ефективними, спершу необхідно усвідомити фундаментальну природу задач оптимізації. У найзагальнішому вигляді задача оптимізації формулюється так: знайти значення змінних $\mathbf{x} = (x_1, x_2, \dots, x_n)$ з деякої допустимої області X, які мінімізують або максимізують цільову функцію $f(\mathbf{x})$. Математично це записується як $\min_{\mathbf{x} \in X} f(\mathbf{x})$ або $\max_{\mathbf{x} \in X} f(\mathbf{x})$.

Здавалося б, задача проста: переберіть усі можливі варіанти і виберіть найкращий. Проте саме тут криється фундаментальна обчислювальна

складність. Для задач комбінаторної оптимізації кількість можливих розв'язків зростає експоненційно або навіть факторіально зі збільшенням розмірності задачі. Наприклад, у класичній задачі комівояжера з n містами існує $(n-1)!/2$ різних маршрутів. Для скромних двадцяти міст це вже понад 10^{17} варіантів - більше, ніж можна перевірити за розумний час навіть на найпотужніших суперкомп'ютерах. Для п'ятдесяти міст кількість варіантів перевищує кількість атомів у видимій частині Всесвіту.

Для неперервних задач ситуація не краща. Якщо простір пошуку містить багато локальних екстремумів, традиційні градієнтні методи неминуче "застрягають" у них, так і не знайшовши глобального оптимуму. Уявімо функцію, що має форму гірського хребта з численними вершинами різної висоти - алгоритм градієнтного підйому, стартувавши з випадкової точки долини, майже напевно зупиниться на найближчій вершині, піднявшись по найкрутішому схилу. Ця вершина може бути далеко не найвищою, але алгоритм не має механізму, щоб спуститися з неї і продовжити пошук у інших регіонах. Більше того, для багатьох реальних задач градієнт взагалі не існує або його важко обчислити - функція може бути дискретною, недиференційовною, зашумленою або обчислюваною лише через складне чисельне моделювання.

Саме для таких "важких" задач оптимізації генетичні алгоритми демонструють свою силу. Вони не гарантують знаходження абсолютного глобального оптимуму за скінченний час, але часто знаходять дуже хороші, практично прийнятні розв'язок там, де традиційні методи оптимізації виявляються безпорадними або вимагають надмірних обчислювальних ресурсів.

Задачі комбінаторної оптимізації

Комбінаторна оптимізація займається задачами, де потрібно знайти найкраще розв'язок з кінцевої, але величезної множини можливих варіантів. Класичним прикладом, який часто використовується для ілюстрації методів оптимізації, є задача комівояжера. У цій задачі комівояжер повинен відвідати n міст, відвідавши кожне місто рівно один раз, і повернутися у початкове місто, пройшовши при цьому мінімальну відстань. Формально, маючи матриця відстаней $D = [d_{ij}]$, де d_{ij} - відстань між містами i та j, потрібно знайти перестановку $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ міст, що мінімізує загальну довжину туру:

$$L(\pi) = \sum_{i=1}^{n-1} d_{\pi_i,\pi_{i+1}} + d_{\pi_n,\pi_1}$$

Чому ця задача така складна? Кількість можливих маршрутів дорівнює $(n-1)!/2$, де ділення на два враховує той факт, що маршрут в прямому та зворотному напрямку має однакову довжину. Для десяти міст це 181,440 варіантів - ще можна перебрати методом грубої сили за секунди. Для двадцяти міст - вже 6×10^{16} варіантів, що вимагає мільйонів років обчислень. Для п'ятдесяти міст повний перебір абсолютно непрактичний. Задача належить до класу NP-важких, що в теорії складності означає відсутність відомих алгоритмів, які б гарантовано знаходили точний оптимум за поліноміальний час для будь-якого екземпляра задачі.

Генетичні алгоритми підходять до цієї задачі принципово інакше. Замість систематичного перебору всіх можливих маршрутів або руху по градієнту вони підтримують популяцію різних маршрутів, поступово покращуючи їх шляхом комбінування частин хороших рішень. Ідея полягає в тому, що якщо один маршрут ефективно з'єднує міста в північній частині карти з мінімальною сумарною відстанню, а інший маршрут має оптимальну послідовність відвідування міст у південній частині, оператор кросинговеру може створити нащадка, який успадкує обидві ці хороші властивості, поєднавши ефективні підмаршрути від обох батьків.

Інша фундаментальна задача комбінаторної оптимізації - це задача про рюкзак. Маємо n предметів, кожен з вагою w_i та цінністю v_i, і рюкзак з обмеженою місткістю W. Потрібно вибрати підмножину предметів так, щоб максимізувати сумарну цінність, не перевищивши при цьому місткість рюкзака. Формально це записується як задача бінарного програмування:

$$\max \sum_{i=1}^{n} v_i \, x_i$$

при обмеженні на вагу:

$$\sum_{i=1}^{n} w_i \, x_i \leq W$$

де $x_i \in \{0,1\}$ - бінарна змінна рішення: $x_i = 1$ означає, що предмет i взято до рюкзака, а $x_i = 0$ означає, що не взято. Розв'язок природно кодується бінарним вектором довжини n, і простір пошуку містить 2^n можливих комбінацій. Для генетичних алгоритмів таке бінарне представлення є

практично ідеальним - воно добре узгоджується з класичним бінарним кодуванням хромосом, запропонованим Холландом.

Задачі складання розкладів також належать до комбінаторних і широко зустрічаються в практиці. Розглянемо, наприклад, складання розкладу занять в університеті. Потрібно призначити кожному заняттю конкретний час, аудиторію та викладача так, щоб не було конфліктів: викладач не може вести дві пари одночасно, студенти однієї групи не можуть бути на двох заняттях одночасно, аудиторія не може бути зайнята кількома групами. Крім того, бажано виконати численні якісні критерії: рівномірність навантаження викладачів по днях тижня, мінімізація "вікон" у розкладі студентів, врахування побажань викладачів щодо часу проведення занять, оптимальне використання спеціалізованих аудиторій. Такі задачі часто мають мільйони або навіть мільярди можливих варіантів, більшість з яких порушують ті чи інші обмеження. Генетичні алгоритми можуть ефективно працювати з частково допустимими рішеннями, використовуючи штрафні функції або спеціальні оператори відновлення допустимості, поступово направляючи пошук до допустимих і оптимальних розкладів.

Задачі неперервної оптимізації

У задачах неперервної оптимізації змінні приймають дійсні значення з деяких діапазонів. Формально задача записується як знаходження мінімуму або максимуму функції багатьох змінних: $\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x})$, де $f\colon \mathbb{R}^n \to \mathbb{R}$ - цільова функція, що відображає n-вимірний простір дійсних чисел у дійсну пряму. Якщо функція f є гладкою, унімодальною (має єдиний екстремум) та диференційовною скрізь, градієнтні методи працюють чудово і швидко збігаються до оптимуму. Класичний метод градієнтного спуску рухається в напрямку антиградієнта, тобто в напрямку найшвидшого зменшення функції. Метод Ньютона використовує інформацію другого порядку (гесіан) для ще швидшої збіжності.

Проте багато реальних задач мають складний багатоекстремальний ландшафт з численними локальними мінімумами, сідловими точками та плато. Розглянемо, наприклад, класичну тестову функцію Растрігіна:

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i)\right)$$

Ця функція має один глобальний мінімум у точці $\mathbf{x} = \mathbf{0}$ зі значенням $f(\mathbf{0}) = 0$, але також має 10^n локальних мінімумів, регулярно розподілених по

всьому простору пошуку з періодом, визначеним косинусоїдальною компонентою. Це створює “хвилясту” поверхню з численними западинами. Градієнтний метод, стартувавши з майже будь-якої початкової точки, потрапить у найближчий локальний мінімум і зупиниться там, будучи не в змозі вибратися назовні, оскільки всі напрямки з локального мінімуму ведуть вгору.

Генетичні алгоритми використовують популяційний підхід, що дозволяє їм одночасно досліджувати різні регіони простору пошуку. Якщо одна частина популяції “застрягла” в одному локальному мінімумі, інші особини можуть досліджувати зовсім інші області простору і потенційно знайти кращі регіони. Крім того, оператор мутації постійно вносить випадкові зміни, що допомагає популяції “вискакувати” з локальних мінімумів, а кросовер дозволяє комбінувати інформацію з різних регіонів простору, створюючи нащадків у точках, які не були явно досліджені батьками.

Важливим класом неперервних задач є оптимізація з обмеженнями. У загальному вигляді така задача записується як:

$$\begin{cases} \min f(\mathbf{x}) \\ g_i(\mathbf{x}) \le 0, \quad i = 1, \dots, m \\ h_j(\mathbf{x}) = 0, \quad j = 1, \dots, p \\ \mathbf{x} \in X \subseteq \mathbb{R}^n \end{cases}$$

Тут g_i представляють обмеження-нерівності, а h_j - обмеження-рівності. У багатьох реальних інженерних та наукових задачах обмеження є нелінійними, і допустима область (множина точок, що задовольняють всі обмеження) може бути незв’язною, мати складну топологію або складатися з кількох ізольованих компонент. Генетичні алгоритми можуть працювати з такими задачами різними способами: використовуючи штрафні функції, що збільшують значення цільової функції пропорційно до величини порушення обмежень; методи відновлення допустимості, що модифікують недопустимі рішення; або спеціальні оператори, що зберігають допустимість при кросинговері та мутації.

Багатокритеріальна оптимізація

У реальному житті рідко зустрічаються задачі з єдиним, чітко визначеним критерієм оптимальності. Зазвичай потрібно одночасно оптимізувати кілька, часто конфліктних між собою цілей. Наприклад, при проектуванні автомобіля інженери хочуть одночасно мінімізувати вартість виробництва, споживання палива, час розгону до ста кілометрів на годину та

максимізувати комфорт пасажирів, безпеку при зіткненні, місткість багажника. Формально багатокритеріальна задача записується як мінімізація векторної функції:

$$\min_{\mathbf{x}\in X}\mathbf{F}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_k(\mathbf{x}))$$

де кожна компонента $f_i(\mathbf{x})$ є окремим критерієм якості.

Фундаментальна проблема багатокритеріальної оптимізації полягає в тому, що рішення, оптимальне за одним критерієм, майже завжди виявляється субоптимальним за іншими критеріями. Якщо ми проектуємо надзвичайно дешевий автомобіль, він швидше за все буде повільним, некомфортним і менш безпечним. Якщо ми проектуємо найшвидший автомобіль, він буде дорогим і споживатиме багато палива. Концепція оптимальності Парето елегантно вирішує цю дилему. Розв'язок $\mathbf{x}^*$ називається Парето-оптимальним або недомінованим, якщо не існує іншого розв'язок $\mathbf{x}$, яке було б не гірше за $\mathbf{x}^*$ за всіма критеріями і строго краще хоча б за одним критерієм. Математично, $\mathbf{x}^*$ є Парето-оптимальним, якщо не існує $\mathbf{x} \in X$ такого, що $f_i(\mathbf{x}) \leq f_i(\mathbf{x}^*)$ для всіх i і $f_j(\mathbf{x}) < f_j(\mathbf{x}^*)$ для деякого j.

Множина всіх Парето-оптимальних розв'язків називається фронтом Парето або множиною Парето. Різні точки на фронті Парето представляють різні компроміси між критеріями. Генетичні алгоритми ідеально підходять для багатокритеріальної оптимізації, оскільки їхня популяційна природа дозволяє знаходити одразу множину різноманітних рішень, що представляють різні компроміси, за один прогін алгоритму. Це принципова перевага над скалярними методами оптимізації, які вимагають багатьох запусків з різними вагами критеріїв для побудови наближення фронту Парето.

Сучасні багатокритеріальні генетичні алгоритми використовують спеціальні механізми для ефективного пошуку та підтримання різноманітності вздовж фронту Парето. Алгоритм NSGA-II використовує концепцію сортування за недомінованістю, де популяція розбивається на фронти: перший фронт містить недоміновані рішення, другий - рішення, недоміновані після видалення першого фронту, і так далі. Всередині кожного фронту підтримується різноманітність через метрику crowding distance, що заохочує рішення, розташовані в менш щільних регіонах фронту. Алгоритм SPEA2 використовує архів елітних розв'язків та

спеціальну фітнес-функцію, що враховує як домінування, так і щільність розв'язків у просторі критеріїв.

Задачі машинного навчання

Генетичні алгоритми знайшли широке та різноманітне застосування в області машинного навчання. Одна з ключових задач - це підбір гіперпараметрів моделі машинного навчання. На відміну від параметрів моделі, які навчаються безпосередньо з даних через оптимізацію функції втрат (наприклад, ваги зв'язків у нейронній мережі навчаються методом зворотного поширення помилки), гіперпараметри встановлюються до початку процесу навчання і визначають структуру та поведінку самого алгоритму навчання. Для нейронних мереж гіперпараметрами можуть бути: кількість прихованих шарів, кількість нейронів у кожному шарі, швидкість навчання, коефіцієнт регуляризації, розмір batch, тип функції активації, параметри dropout тощо.

Простір гіперпараметрів часто є змішаним, тобто містить як дискретні, так і неперервні змінні різних типів і масштабів. Наприклад, кількість шарів - це дискретне ціле число, швидкість навчання - дійсне число, яке зазвичай варіюється в логарифмічному масштабі, а тип оптимізатора - категоріальна змінна з обмеженого набору варіантів. Цей простір не має чіткої структури, яку можна було б використати для градієнтного пошуку або аналітичної оптимізації. Більше того, функція якості (наприклад, точність класифікації на валідаційній вибірці) як функція гіперпараметрів є зашумленою, недиференційовною і може мати складну багатоекстремальну структуру. Генетичні алгоритми не роблять припущень про структуру або властивості простору пошуку і можуть ефективно працювати з такими гетерогенними задачами.

Інша важлива задача машинного навчання, де генетичні алгоритми демонструють переваги, - це відбір ознак або feature selection. При наявності великої кількості потенційних ознак для побудови моделі машинного навчання постає питання: які ознаки дійсно корисні для передбачення цільової змінної, а які є шумом або надлишковими? Задача полягає у виборі підмножини з n доступних ознак, яка забезпечує найкращу якість класифікації або регресії при мінімальній складності моделі. Це комбінаторна задача з 2^n можливими варіантами підмножин. Розв'язок природно кодується бінарним вектором $\mathbf{x} = (x_1, \dots, x_n)$, де $x_i = 1$ означає, що ознака i включена до набору, а $x_i = 0$ означає, що вона виключена. Функція

пристосованості зазвичай поєднує точність моделі, навченої на відібраному наборі ознак, з кількістю відібраних ознак:

$$f(\mathbf{x}) = \alpha \cdot \text{Accuracy}(\mathbf{x}) - \beta \cdot \frac{\sum_{i=1}^{n} x_i}{n}$$

де α та β - вагові коефіцієнти, що визначають компроміс між точністю та простотою моделі. Генетичний алгоритм досліджує простір можливих підмножин ознак, знаходячи комбінації, що забезпечують високу точність при невеликій кількості ознак.

Коли генетичні алгоритми не є оптимальним вибором

Важливо розуміти не тільки переваги, а й обмеження методу. Генетичні алгоритми не є універсальною панацеєю і мають свої принципові недоліки та ситуації, де інші методи значно ефективніші. По-перше, генетичні алгоритми не гарантують знаходження глобального оптимуму за скінченний або навіть розумний час і часто вимагають багато обчислень функції пристосованості. Якщо кожне обчислення функції займає години або дні (наприклад, складне чисельне моделювання фізичної системи, кліматична модель або детальне комп'ютерне моделювання методом скінченних елементів), генетичний алгоритм може виявитися непрактичним через величезний обчислювальний бюджет, необхідний для еволюції популяції протягом багатьох поколінь.

По-друге, для задач з гладкими, унімодальними функціями без обмежень або з простими лінійними обмеженнями класичні методи оптимізації працюють набагато швидше і точніше. Якщо ви оптимізуєте квадратичну функцію, метод Ньютона або метод спряжених градієнтів знайде точний оптимум за кілька ітерацій, вимагаючи десятки обчислень функції та її похідних. Генетичному алгоритму знадобляться тисячі або десятки тисяч обчислень функції для досягнення навіть наближеної точності, і він ніколи не досягне машинної точності в силу стохастичної природи.

По-третє, для багатьох класів задач існують спеціалізовані алгоритми, які використовують специфічну структуру або властивості задачі і працюють на порядки ефективніше за будь-який загальний метод оптимізації. Наприклад, для задачі про максимальний потік у мережі алгоритм Форда-Фалкерсона або його модифікації знаходять точний оптимум за поліноміальний час. Для задач лінійного програмування симплекс-метод або методи внутрішньої точки гарантовано збігаються до оптимуму. Для задач на графах існують ефективні алгоритми динамічного програмування,

жадібні алгоритми або алгоритми пошуку шляхів, які використовують структуру графа.

Генетичні алгоритми показують найкращі результати і є методом вибору саме там, де: простір пошуку є величезним і складно структурованим; функція пристосованості має багато локальних екстремумів; відсутня аналітична інформація про градієнти або функція взагалі недиференційовна; потрібна стійкість до шуму в обчислюваній функції якості; важлива є знаходження не одного, а множини хороших різноманітних рішень; задача є багатокритеріальною з конфліктними цілями; немає спеціалізованих алгоритмів для даного класу задач або вони недостатньо ефективні на практиці.

3. Біологічна аналогія та головна ідея генетичних алгоритмів

Щоб глибоко зрозуміти принцип роботи генетичних алгоритмів на концептуальному рівні, корисно спершу детально розглянути, як працює еволюція в природі. Теорія еволюції Чарльза Дарвіна, сформульована в середині дев'ятнадцятого століття в монументальній праці "Походження видів шляхом природного добору", пояснює, як з простих одноклітинних організмів протягом мільярдів років виникло величезне різноманіття складних форм життя, що населяють нашу планету.

У природі еволюція відбувається через взаємодію кількох ключових механізмів, що діють одночасно та взаємопов'язано. Першим фундаментальним механізмом є спадковість - здатність організмів передавати свої характеристики нащадкам через генетичний матеріал. У більшості живих організмів генетична інформація закодована в молекулах дезоксирибонуклеїнової кислоти, відомої як ДНК. Ця інформація організована в послідовності нуклеотидів чотирьох типів (аденін, тимін, гуанін, цитозін), які можна розглядати як чотирьохбуквовий алфавіт життя. Окремі функціональні одиниці спадковості називаються генами, і кожен ген зазвичай кодує певну ознаку організму або відповідає за синтез певного білка.

Другим критично важливим механізмом є мінливість - поява нових варіантів генів та нових комбінацій існуючих генів. Мінливість виникає через два основні процеси. Перший - це мутації, випадкові зміни в послідовності ДНК, що можуть відбуватися через помилки копіювання при реплікації ДНК, вплив іонізуючого випромінювання, хімічних мутагенів або просто через статистичні флуктуації на молекулярному рівні. Більшість

мутацій є нейтральними або навіть шкідливими для організму, але час від часу виникає корисна мутація, що покращує пристосованість організму до середовища. Другий джерело мінливості - це рекомбінація або схрещування, обмін генетичним матеріалом між батьками при статевому розмноженні. Під час утворення статевих клітин відбувається процес, званий кросинговером, коли гомологічні хромосоми обмінюються ділянками, створюючи нові комбінації генів, яких не було у жодного з батьків.

Третім і, можливо, найважливішим механізмом є природний відбір - диференційне виживання та розмноження організмів залежно від їхньої пристосованості до навколишнього середовища. Організми з генами, які краще пристосовують їх до конкретних умов існування, мають статистично вищі шанси вижити до репродуктивного віку, знайти партнера і залишити нащадків. Ці нащадки успадковують корисні гени, які, в свою чергу, мають вищу ймовірність передатися наступному поколінню. Через багато поколінь корисні гени поширюються в популяції, їхня частота зростає, тоді як шкідливі генетичні варіанти поступово елімінуються з генофонду. Це і є суть природного відбору - не випадкові зміни самі по собі, а направлений, хоч і ненавмисний процес збереження та накопичення корисних варіацій.

Критично важливою для розуміння еволюції є її популяційна природа. Еволюція відбувається не з окремими організмами, які народжуються з одним генотипом і помирають з ним же, а з популяціями - групами організмів одного виду, які можуть схрещуватися між собою. Генетична різноманітність популяції, тобто наявність різних алелів (варіантів) генів у різних особин, є тим матеріалом, з яким працює природний відбір. Популяція без генетичної різноманітності, де всі особини генетично ідентичні, не може еволюціонувати, оскільки немає варіацій для відбору. Більше того, генетична різноманітність забезпечує стійкість популяції до змін середовища - навіть якщо умови раптово змінюються, деякі генетичні варіанти, що раніше були нейтральними або навіть трохи шкідливими, можуть виявитися корисними в нових умовах.

Джон Холланд побачив глибоку математичну аналогію між цим процесом природної еволюції та процесом пошуку оптимальних розв’язків у просторі можливих варіантів. Він усвідомив, що задачу оптимізації можна переформулювати в еволюційних термінах: замість пошуку оптимуму функції ми можемо мислити про адаптацію популяції розв’язків до “середовища”, визначеного функцією пристосованості. Розглянемо цю

аналогію більш детально, встановлюючи відповідності між біологічними поняттями та математичними конструкціями.

Організм у біології відповідає кандидату розв'язок або точці в просторі пошуку в оптимізації. Хромосома, фізичний носій генетичної інформації в клітині, відповідає кодуванню або представленню розв'язок - часто це бітовий рядок, вектор дійсних чисел або перестановка, залежно від природи задачі. Ген, функціональна одиниця спадковості, що займає певну позицію на хромосомі, відповідає окремій змінній розв'язок або параметру, що оптимізується. Алель, конкретний варіант гену, відповідає конкретному значенню цієї змінної.

Генотип - повний набір генів організму - це внутрішнє представлення рішення, кодування, з яким працює алгоритм. Фенотип - сукупність зовнішніх ознак організму, що виникають в результаті взаємодії генотипу та середовища - це декодоване рішення, яке насправді оцінюється функцією пристосованості. Ця різниця між генотипом та фенотипом важлива: в біології один і той же генотип може давати різні фенотипи в різних умовах, а в генетичних алгоритмах різні внутрішні представлення можуть після декодування давати однакове рішення.

Пристосованість організму в біології, його здатність виживати та розмножуватися в даному середовищі, прямо відповідає якості рішення, значенню функції оптимізації. Середовище, сукупність зовнішніх умов, до яких організм повинен адаптуватися, відповідає задачі оптимізації, яку ми намагаємося розв'язати. Покоління, зміна популяції організмів через цикл розмноження та смерті, відповідає ітерації алгоритму, де стара популяція розв'язків замінюється новою.

Ключова ідея Холланда полягала в тому, що процес еволюції можна розглядати як надзвичайно ефективний адаптивний метод пошуку в величезних просторах можливостей. Природний відбір "пробує" різні генетичні комбінації в буквальному сенсі - організми з різними генами народжуються, живуть і або успішно розмножуються, або гинуть без нащадків. Середовище "оцінює" ці комбінації через виживання та репродуктивний успіх. Успішні комбінації зберігаються та розмножуються, неуспішні - зникають. Аналогічно, генетичний алгоритм пробує різні розв'язок задачі оптимізації, оцінюючи їх функцією пристосованості, і зберігає кращі з них, використовуючи їх як основу для генерації нових, потенційно ще кращих рішень.

Архітектура життєвого циклу генетичного алгоритму

Генетичний алгоритм функціонує як ітераційний цикл, що моделює еволюційний процес. Цей цикл повторюється до виконання певного критерію зупинки.

Етап 1: Кодування задачі (представлення хромосом)

Першим критичним кроком є кодування потенційного розв’язок у формат, придатний для генетичних операторів. Розв’язок має бути представлено у вигляді хромосоми.[2] Наприклад, для простих оптимізаційних задач може використовуватися бінарний рядок, але для задач з обмеженнями, як-от Задача Комівояжера (де елементи не можуть повторюватися), необхідне кодування на основі перестановки.[5] Успішність ГА значною мірою залежить від того, наскільки ефективно обрана схема кодування дозволяє генетичним операторам комбінувати та змінювати елементи, зберігаючи при цьому валідність рішення.

Етап 2: Ініціалізація початкової популяції

На цьому етапі випадковим чином створюється деяка кількість початкових елементів («осіб»).

Рандомізація є ключовою для забезпечення первісного дослідження простору пошуку. Створення достатньо різноманітної початкової популяції гарантує, що алгоритм розпочне пошук у різних областях простору рішень, навіть якщо початкові особини виявляться абсолютно неконкурентоздатними.

Етап 3: Обчислення функції придатності та оцінювання

Після ініціалізації кожна особа в популяції має бути оцінена. Це здійснюється за допомогою функції придатності. Функція придатності визначає, наскільки якісним є кожне рішення. У результаті оцінювання кожній особі присвоюється певне числове значення, яке безпосередньо визначає її можливість виживання та участі у розмноженні. Таким чином, функція придатності є мірою успішності особини у розв'язанні поставленої задачі, і це значення стає критерієм для селекції.

Етап 4: Повторення та Еволюційні Оператори

Основний цикл ГА включає послідовність еволюційних операторів, які повторюються до виконання критерію зупинки:

1. Вибір (Селекція): Відбираються індивіди з поточної популяції, які мають високі значення придатності.
2. Схрещування (Кросовер): Обрані особи (батьки) обмінюються генетичним матеріалом для створення нового покоління нащадків.
3. Мутація: До генетичного матеріалу нащадків вносяться випадкові зміни, що запобігає стагнації.
4. Формування нового покоління: Оцінюється функція пристосованості для всіх нових осіб, і вони заміщують або доповнюють стару популяцію.

Різноманіття хромосом, яке підтримується цими операторами, є ключовим елементом, що дозволяє алгоритму ефективно виходити з локальних мінімумів і знаходити оптимальні розв’язок в усьому просторі пошуку.

Умова зупинки алгоритму

Цикл еволюції триває до моменту, коли буде виконано один із заздалегідь визначених критеріїв зупинки. Основні критерії зупинки включають:

1. Знаходження глобального або надоптимального рішення, яке відповідає заданому порогу якості.
2. Вичерпання фіксованого числа поколінь, що були відпущені на еволюцію.
3. Вичерпання часу, відпущеного на обчислення.

Стагнація, тобто відсутність значного покращення найкращого розв’язок протягом певної кількості послідовних поколінь.

Як ми бачимо, концептуальна схема генетичного алгоритму дуже проста і елегантна у своїй простоті. Але все ж таки ще раз детальніше її розглянемо.

Алгоритм починає з популяції випадково згенерованих рішень-кандидатів. Ця початкова популяція, як правило, далека від оптимуму - це просто випадкові точки в просторі пошуку. Проте ця популяція містить важливий ресурс: різноманітність. Різні особини знаходяться в різних регіонах простору пошуку, несуть різну генетичну інформацію. Далі запускається

ітераційний процес еволюції, де кожна ітерація називається поколінням або генерацією.

На кожній ітерації відбувається послідовність кроків, що імітує природну еволюцію. Спочатку всі розв'язок в популяції оцінюються: для кожного розв'язок обчислюється функція пристосованості, що вимірює його якість. Це аналог того, як середовище "оцінює" організм - наскільки добре він пристосований до умов існування. На основі цих оцінок відбувається селекція: вибираються батьківські особини для створення нащадків. Кращі рішення, з вищою пристосованістю, мають вищу ймовірність бути вибраними як батьки, що моделює природний відбір. Важливо розуміти, що селекція є стохастичною, імовірнісною - навіть менш пристосовані особини мають шанс бути вибраними, хоч і меншу ймовірність. Це критично важливо для підтримання генетичної різноманітності популяції.

Після селекції відбувається кросовер або схрещування: пари батьківських розв'язків комбінуються для створення нащадків. Частини генетичного матеріалу від одного батька поєднуються з частинами від іншого, створюючи нові комбінації генів. Це моделює статеве розмноження та генетичну рекомбінацію в природі. Фундаментальна ідея полягає в тому, що якщо два розв'язок мають хороші, але різні властивості, їхня комбінація може успадкувати переваги обох батьків, створюючи нащадка, кращого за обох. Наприклад, якщо в задачі комівояжера один батьківський тур ефективно з'єднує міста в північній частині карти, а другий - в південній, нащадок може успадкувати обидва ці хороші підмаршрути.

Потім до нащадків застосовується мутація: з певною невеликою ймовірністю кожен ген випадково змінюється. Це моделює генетичні мутації в природі і служить двом важливим цілям. По-перше, мутація підтримує генетичну різноманітність, запобігаючи передчасній збіжності популяції до одного рішення. По-друге, мутація вводить новий генетичний матеріал, якого не було в початковій популяції - вона дозволяє алгоритму досліджувати регіони простору пошуку, які не були представлені в початковій випадковій популяції.

Нарешті, відбувається заміна або формування нової популяції: новостворені нащадки замінюють частину або всю стару популяцію, формуючи популяцію наступного покоління. Існують різні стратегії заміни. При повній заміні вся стара популяція замінюється нащадками. При елітизмі кілька найкращих особин з поточної популяції гарантовано переходять у наступне покоління без змін, що забезпечує монотонність

покращення найкращого рішення. При steady-state заміні лише кілька найгірших особин замінюються нащадками кожної ітерації.

Цей цикл - оцінювання, селекція, кросинговер, мутація, заміна - повторюється багато разів, від десятків до тисяч поколінь залежно від складності задачі. З кожним поколінням середня та найкраща пристосованість популяції зазвичай зростає, хоча можливі флуктуації через стохастичну природу операторів. Алгоритм зупиняється при виконанні деякого критерію: досягнення заданої якості рішення, вичерпання обчислювального бюджету, завершення відведеного часу або стагнації, коли протягом багатьох поколінь не відбувається помітного покращення.

Ефективність генетичних алгоритмів може здаватися дивною на перший погляд. Адже вони використовують випадкові зміни - кросовер випадково комбінує частини батьків, мутація випадково змінює гени. Як може випадковість призводити до систематичного покращення? Ключ у поєднанні випадковості з направленим відбором. Випадкові зміни генерують різноманітність, створюють нові варіанти для перевірки. Селекція надає пошуку напрямленість, концентруючи зусилля на перспективних регіонах простору. Разом вони створюють потужний механізм адаптивного пошуку.

Уявімо простір пошуку як складний багатовимірний ландшафт, де висота в кожній точці відповідає якості рішення. Завдання оптимізації - знайти найвищу вершину цього ландшафту, глобальний максимум функції пристосованості. Популяція генетичного алгоритму - це група дослідників, розкиданих по цьому ландшафту. Кожен дослідник знаходиться в якійсь точці і знає висоту в цій точці, але не має карти всього ландшафту і не знає, де саме він знаходиться відносно глобального максимуму.

Кросовер діє як механізм обміну інформацією між дослідниками. Якщо один дослідник знайшов високу точку на півночі ландшафту, а інший - високу точку на сході, кросовер створює нового дослідника, який спробує дослідити північно-східну область. Логіка проста: якщо обидва батьки знайшли хороші регіони в різних напрямках, можливо, комбінація цих напрямків також буде хорошою. Звичайно, це не завжди так - ландшафт може бути хитрим, і північно-східна область може виявитися западиною. Але статистично, якщо батьки дійсно хороші, є шанс, що їхня комбінація також буде хорошою.

Мутація діє як механізм випадкових стрибків. Час від часу дослідник робить випадковий стрибок у сусідню область або навіть в зовсім віддалений регіон ландшафту. Це може здаватися нерозумним - навіщо стрибати випадково, коли ми хочемо рухатися вгору? Причина в тому, що дослідник може застрягти на локальній вершині, оточеній долинами. Всі сусідні точки нижчі, і детермінований локальний пошук зупиниться. Мутація дає шанс перестрибнути через долину на схил іншої, можливо вищої вершини.

Селекція концентрує зусилля популяції на перспективних напрямках. Якщо дослідники на північ знайшли в середньому вищі точки, ніж дослідники на південь, майбутні покоління будуть більше досліджувати північні області. Це не означає повної відмови від південних областей - деякі дослідники залишаться там на випадок, якщо там є щось цікаве. Але ресурси популяції (кількість особин, що досліджують регіон) динамічно перерозподіляються на користь більш обіцяючих регіонів.

Важливо розуміти, що генетичний алгоритм не просто випадково блукає простором пошуку і не просто детерміновано рухається по градієнту. Це складна взаємодія дослідження нових регіонів і експлуатації знайдених хороших областей. На початку еволюції, коли популяція різноманітна і ще не збіглася, домінує дослідження - алгоритм пробує різні регіони простору. Пізніше, коли популяція починає збігатися навколо хороших рішень, домінує експлуатація - тонке налаштування знайдених рішень. Баланс між дослідженням та експлуатацією контролюється параметрами алгоритму: розміром популяції, ймовірностями кросинговеру та мутації, селекційним тиском.

Холланд не обмежився інтуїтивними міркуваннями та аналогіями. Він створив строгу математичну теорію, що пояснює ефективність генетичних алгоритмів. Центральним елементом цієї теорії є концепція схем та теорема схем, яку часто називають фундаментальною теоремою генетичних алгоритмів.

Схема - це шаблон або паттерн, що описує підмножину рішень. Для бінарних рядків довжини l схема записується з використанням алфавіту $\{0,1,*\}$, де зірочка означає “будь-яке значення”, тобто може бути як 0, так і 1.

Наприклад, схема $*1*0*$ для рядків довжини п’ять описує всі рядки, у яких друга позиція обов’язково дорівнює одиниці, а четверта - нулю, тоді як інші позиції можуть бути будь-якими. Конкретно, ця схема включає вісім рядків:

01000, 01001, 01100, 01101, 11000, 11001, 11100, 11101.

Можна думати про схему як про узагальнене розв'язок або будівельний блок - це частковий опис, що фіксує деякі важливі властивості множини рішень.

Для кожної схеми можна визначити дві важливі характеристики, що впливають на її поведінку в генетичному алгоритмі.

Означення 1. Порядок схеми $o(H)$ - це кількість фіксованих позицій, тобто кількість позицій, в яких стоїть не зірочка.

Для схеми $*1*0*$ порядок дорівнює двом, оскільки зафіксовано дві позиції.

Означення 2. Довжина визначення схеми $\delta(H)$ - це відстань між першою та останньою фіксованою позицією.

Для $*1*0*$ перша фіксована позиція - друга, остання фіксована - четверта, тому довжина визначення дорівнює $4-2=2$.

Теорема схем Холланда робить фундаментальну заяву про динаміку схем у генетичному алгоритмі.

Теорема схем (Холланда). Схеми з низьким порядком, короткою довжиною визначення та пристосованістю вище середньої експоненційно зростають у популяції в наступних поколіннях. Математично, очікувана кількість представників схеми H у поколінні $t+1$ оцінюється знизу як:

$$E[m(H,t+1)] \geq m(H,t) \cdot \frac{f(H)}{f_{avg}} \cdot \left[1 - p_c \cdot \frac{\delta(H)}{l-1} - o(H) \cdot p_m\right]$$

Тут $m(H,t)$ позначає кількість екземплярів схеми H у популяції в поколінні t.

Величина $f(H)$ - це середня пристосованість всіх розв'язків у популяції, що відповідають схемі H.

Величина f_{avg} - середня пристосованість усієї популяції.

Параметр p_c - ймовірність застосування кросинговеру до пари батьків,

Параметр p_m - ймовірність мутації одного біта,

Параметр l - загальна довжина бінарного рядка.

Розглянемо, що говорить нам кожен множник цієї нерівності.

Перший множник $\frac{f(H)}{f_{avg}}$ показує селекційний ефект. Якщо середня пристосованість схеми більша за середню пристосованість популяції, тобто

$f(H) > f_{avg}$, то цей множник більший за одиницю, і схема зростає через селекцію. Рішення, що відповідають хорошій схемі, частіше вибираються як батьки, і тому схема поширюється. Якщо схема погана, $f(H) < f_{avg}$, множник менший за одиницю, і схема зменшується.

Вираз у квадратних дужках описує деструктивний ефект генетичних операторів.

Член $p_c \cdot \frac{\delta(H)}{l-1}$ оцінює ймовірність руйнування схеми кросинговером. Якщо довжина визначення схеми велика, тобто фіксовані позиції розкидані по всьому рядку, ймовірність, що точка розрізу кросинговеру потрапить між ними і розірве схему, висока. Навпаки, якщо схема коротка, $\delta(H)$ мале, фіксовані біти розташовані близько один до одного, і кросовер скоріше за все не розірве їх - вони перейдуть разом до нащадка.

Член $o(H) \cdot p_m$ оцінює ймовірність руйнування схеми мутацією. Кожна фіксована позиція може мутувати з ймовірністю p_m, і якщо порядок схеми високий, є багато позицій, які можуть зіпсуватися.

Об'єднуючи ці ефекти, теорема схем говорить:

короткі схеми низького порядку з високою пристосованістю експоненційно зростають у популяції.

Холланд назвав такі схеми "будівельними блоками". Генетичний алгоритм ефективний, оскільки він одночасно ідентифікує хороші будівельні блоки в різних регіонах простору пошуку і комбінує їх через кросинговер, створюючи рішення, що містять кілька хороших будівельних блоків одночасно.

Ще важливішим є висновок про неявний паралелізм. Популяція розміру N, що складається з бінарних рядків довжини l, містить $N \cdot l$ біт явної інформації. Але скільки різних схем представлено в цій популяції? Кожен рядок довжини l є екземпляром 2^l різних схем - для кожної підмножини позицій можна сформувати схему, що фіксує ці позиції і залишає решту вільними.

Але багато схем представлені в кількох різних рядках. Холланд показав, що типова популяція неявно обробляє приблизно $O(N^3)$ різних схем паралельно. Це означає, що генетичний алгоритм, явно працюючи лише з N індивідуальними рішеннями (типово N близько ста), насправді "оцінює" та "відбирає" кубічно більшу кількість схем - підмножин простору пошуку.

Це надзвичайно потужна властивість: алгоритм досліджує величезну кількість підобластей простору, не обчислюючи явно функцію для кожної з них.

4. Кодування розв'язків у генетичних алгоритмах

Перш ніж застосовувати генетичний алгоритм до конкретної задачі, необхідно вирішити фундаментальне питання представлення: як закодувати розв'язок задачі у вигляді, придатному для генетичних операторів? Це питання кодування або представлення є критично важливим і часто визначає успіх або невдачу застосування генетичного алгоритму. Вибір кодування впливає на складність операторів, ефективність пошуку, збіжність алгоритму та якість знайдених рішень.

Історично першим і найбільш вивченим представленням є бінарне кодування, запропоноване Джоном Холландом. У цьому підході кожне розв'язок представлено у вигляді бінарного рядка або бітового вектора фіксованої довжини:

$\mathbf{x} = (x_1, x_2, \dots, x_l)$,

де кожен $x_i \in \{0,1\}$.

Для задач дискретної оптимізації, де змінні природно є бінарними (так/ні, включено/виключено, взяти/не взяти), таке представлення безпосередньо відображає структуру задачі. Наприклад, у задачі про рюкзак кожен біт вказує, чи включено відповідний предмет до рюкзака.

Для задач неперервної оптимізації, де змінні приймають дійсні значення з певних діапазонів, бінарне кодування вимагає процедури дискретизації.

Припустимо, потрібно представити дійсне число y з діапазону $[a, b]$ з заданою точністю. Виберемо кількість бітів k для кодування цієї змінної. Ці k бітів можуть представляти 2^k різних цілих чисел від 0 до $2^k - 1$.

Відображення від бінарного рядка до дійсного числа здійснюється за формулою:

$$y = a + \frac{b - a}{2^k - 1} \cdot \sum_{i=1}^{k} x_i \cdot 2^{k-i}$$

Тут $\sum_{i=1}^{k} x_i \cdot 2^{k-i}$ - це просто десяткове значення k-бітного двійкового числа.

Множник $\frac{b-a}{2^k-1}$ масштабує діапазон від $[0, 2^k - 1]$ до $[a, b]$.

Точність представлення визначається кількістю бітів: чим більше k, тим дрібніша сітка дискретизації і тим точніше можна представити дійсне число.

Наприклад, нехай потрібно закодувати змінну $y \in [-5,5]$ з точністю принаймні 0.01. Діапазон має довжину 10, тому потрібно принаймні $10/0.01 = 1000$ різних значень. Оскільки

$2^9 = 512 < 1000 < 1024 = 2^{10}$,

потрібно $k = 10$ бітів.

Бінарний рядок 0000000000 (всі нулі)

буде відповідати $y = -5$,

рядок 1111111111 (всі одиниці)

– значенню $y = -5 + 10 \cdot \frac{1023}{1023} = 5$,

а рядок 1000000000 (перший біт одиниця, решта нулі)

- значенню $y = -5 + 10 \cdot \frac{512}{1023} \approx -0.002$.

Якщо задача має кілька дійсних змінних, кожна кодується своїм набором бітів, і всі ці набори конкатенуються в один довгий бінарний рядок.

Наприклад, для функції двох змінних $f(x_1, x_2)$, де $x_1 \in [0,10]$ і $x_2 \in [-5,5]$, можна використати по 10 бітів на кожну змінну, отримуючи хромосому довжини 20 бітів.

Бінарне кодування має кілька переваг. Воно є простим концептуально і легко реалізується. Генетичні оператори - одноточковий кросинговер, мутація через інверсію біта - тривіальні для реалізації. Існує потужна теорія, включаючи теорему схем Холланда, розроблена саме для бінарного представлення. Бінарні рядки можна розглядати як універсальне представлення, оскільки будь-яку інформацію можна закодувати в бінарному вигляді.

Проте бінарне кодування має і серйозні недоліки. Головна проблема - це феномен Гемінгової скелі. Гемінгова відстань між двома бінарними рядками - це кількість позицій, в яких вони відрізняються. У стандартному бінарному кодуванні сусідні числа можуть мати велику Гемінгову відстань.

Наприклад, числа 7 і 8 в чотирибітному коду представлені як 0111 і 1000 - вони відрізняються у всіх чотирьох бітах! Це означає, що невелика зміна в числовому значенні вимагає одночасної зміни багатьох бітів, що

малоймовірно при випадковій мутації. Це порушує принцип локальності: близькі розв'язок повинні мати близькі представлення.

Для вирозв'язок проблеми Гемінгової скелі було запропоновано кодування Грея, названого на честь дослідника Френка Грея з Bell Labs. Код Грея - це альтернативний спосіб представлення цілих чисел в бінарному вигляді з надзвичайно важливою властивістю: послідовні числа в коді Грея відрізняються рівно в одному біті. Інакше кажучи, Гемінгова відстань між представленнями чисел n і $n + 1$ у коді Грея завжди дорівнює одиниці.

Розглянемо чотирибітні коди для чисел від 0 до 15:

Число	Стандартний бінарний	Код Грея
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Звернімо увагу на перехід від 7 до 8 у стандартному коді:

0111 → 1000 - чотири біти змінилися.

У коді Грея:

0100 → 1100 - змінився лише один біт (перший).

Ця властивість зберігається для всіх послідовних пар.

Перетворення від стандартного бінарного коду до коду Грея та назад здійснюється простими бітовими операціями. Для перетворення k-бітного бінарного числа

$\mathbf{b} = (b_1, b_2, \dots, b_k)$

в код Грея

$\mathbf{g} = (g_1, g_2, \dots, g_k)$

використовується формула:

$$g_1 = b_1$$
$$g_i = b_{i-1} \oplus b_i, \quad i = 2, \dots, k$$

де $\oplus$ - операція XOR (виключне АБО). Фактично, кожен біт коду Грея є XOR двох сусідніх бітів бінарного коду.

Зворотне перетворення від коду Грея до стандартного бінарного:

$$b_1 = g_1$$
$$b_i = b_{i-1} \oplus g_i, \quad i = 2, \dots, k$$

Тут кожен біт бінарного коду обчислюється як XOR попереднього біта бінарного коду і поточного біта коду Грея.

Властивість послідовності означає, що мутація одного біта в коді Грея призводить до невеликої зміни числового значення - перехід до одного з сусідніх чисел у числовому ряді. Це встановлює локальність: малі зміни в генотипі (представленні) призводять до малих змін у фенотипі (значенні). Це узгоджується з принципом поступової адаптації: еволюція працює через невеликі кроки покращення, і представлення повинно підтримувати цей процес.

Емпіричні дослідження показують, що для функцій неперервної оптимізації код Грея часто забезпечує швидшу збіжність генетичного алгоритму і кращу якість кінцевих розв’язків порівняно зі стандартним бінарним кодуванням. Особливо значною є перевага на багатомодальних функціях з великою кількістю локальних мінімумів, де здатність робити малі кроки критична для тонкої настройки рішень.

Проте навіть код Грея не вирішує всіх проблем дискретного представлення неперервних змінних. Точність обмежена кількістю бітів, що може бути недостатньою для деяких задач. Оператори кросинговеру можуть створювати нащадків, значення яких не лежать між батьківськими значеннями, що іноді небажано. Ці міркування призвели до розвитку альтернативних схем кодування.

Дійсне кодування представляє розв’язок безпосередньо як вектор дійсних чисел:

$\mathbf{x} = (x_1, x_2, \dots, x_n)$,

де кожне $x_i \in \mathbb{R}$ або $x_i \in [a_i, b_i]$ для деякого допустимого діапазону.

Це природне представлення для задач неперервної оптимізації - немає необхідності в кодуванні та декодуванні, розв'язок вже представлене в тих самих термінах, що й задача.

Дійсне кодування усуває проблеми дискретизації та обмеженої точності бінарного представлення. Розв'язок може бути представлено з машинною точністю (зазвичай 64-бітні числа з плаваючою точкою дають точність близько 10^{-16}). Відпадає необхідність вибирати кількість бітів на змінну. Проте дійсне кодування вимагає спеціалізованих генетичних операторів, оскільки бітові операції не мають сенсу для дійсних чисел.

Перестановочне кодування використовується для задач, де розв'язок є впорядкуванням або послідовністю елементів.

Розв'язок представлено як перестановка чисел від 1 до n: $\mathbf{x} = (\pi_1, \pi_2, \dots, \pi_n)$, де кожне число від 1 до n зустрічається рівно один раз.

Класичний приклад - задача комівояжера, де перестановка визначає порядок відвідування міст. Наприклад, (3,1,4,2,5) означає маршрут: починаємо з міста 3, потім йдемо в місто 1, далі в 4, потім у 2, потім у 5, і нарешті повертаємося в початкове місто 3.

Перестановочне кодування критично вимагає спеціалізованих операторів. Стандартний кросовер або мутація для бінарних або дійсних представлень може створити недопустиму перестановку, де деякі елементи дублюються, а інші відсутні. Необхідні оператори, що гарантують збереження властивості перестановки.

Деревоподібне кодування представляє розв'язок як дерево, зазвичай синтаксичне дерево програми або математичного виразу. Це основа генетичного програмування, розробленого Джоном Козою. Внутрішні вузли дерева - функції або оператори, листя - термінальні символи (змінні, константи).

Наприклад, арифметичний вираз $(x + 3) \cdot (y - 2)$ представляється деревом з коренем-операцією множення, лівим піддеревом для $(x + 3)$ і правим для $(y - 2)$.

Матричне або структурне кодування використовується для задач з більш складною структурою. Наприклад, у задачах проектування нейронних мереж архітектура може кодуватися як матриця суміжності графа з'єднань

або як ієрархічна структура блоків. У задачах проектування електронних схем топологія може бути закодована як граф компонентів та з'єднань.

Вибір правильного кодування - це мистецтво, що вимагає глибокого розуміння як задачі, так і алгоритму. Хороше кодування повинно задовольняти кілька принципів.

По-перше, повнота: кожне допустиме розв'язок задачі повинно мати принаймні одне представлення в обраному кодуванні. Якщо деякі потенційно оптимальні розв'язок не можуть бути представлені, алгоритм ніколи їх не знайде.

По-друге, обґрунтованість: кожне можливе представлення (хромосома) повинно відповідати допустимому рішенню задачі, або принаймні має існувати процедура відновлення допустимості. Якщо багато хромосом є недопустимими, алгоритм витрачатиме час на обробку безглуздих варіантів.

По-третє, локальність або природність: близькі представлення (в сенсі Гемінгової відстані для бінарних, евклідової для дійсних, відстані редагування для перестановок) повинні відповідати близьким рішенням у сенсі якості або властивостей. Це забезпечує ефективність мутації як локального пошуку і дозволяє кросинговеру осмисленно комбінувати властивості батьків.

По-четверте, лаконічність: представлення не повинно бути надлишковим. Якщо багато різних хромосом кодують одне і те ж рішення, пошуковий простір штучно роздутий, і алгоритм може марнувати час, знаходячи різні представлення одного і того ж розв'язок замість дослідження нових областей.

5. Детальний огляд генетичних операторів

Генетичні оператори є серцем еволюційного процесу. Вони визначають, як популяція трансформується від покоління до покоління, як комбінуються хороші рішення, як вводиться різноманітність, як балансується дослідження нових регіонів і експлуатація знайдених хороших областей. Розглянемо детально основні класи операторів з теоретичними обґрунтуваннями, прикладами застосування та псевдокодом реалізації.

Оператори селекції

Селекція моделює природний відбір, вибираючи особини для розмноження на основі їхньої пристосованості. Математично, селекція - це стохастична

функція, що відображає поточну популяцію $P(t) = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ з оцінками пристосованості $\{f_1, f_2, \dots, f_N\}$ у батьківську множину P_{parent} того ж або меншого розміру.

Пропорційна селекція

У пропорційній селекції ймовірність вибору особини прямо пропорційна її пристосованості відносно сумарної пристосованості популяції. Ймовірність вибору i-ої особини:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

Геометрично можна уявити рулетку, де кожна особина займає сектор, пропорційний її пристосованості. Обертання рулетки з випадковою зупинкою відповідає вибору одного індивіда. Процес повторюється N разів для формування батьківської популяції.

Теоретично, це означає, що особина з пристосованістю вдвічі більшою за середню має очікувану кількість копій у батьківській популяції рівну двом. Особина з середньою пристосованістю має очікувану кількість копій рівну одиниці. Це створює селекційний тиск: кращі розв'язок розмножуються, гірші зникають.

Проблеми пропорційної селекції включають передчасну збіжність, коли одна надзвичайно хороша особина на ранніх етапах домінує популяцію, знищуючи різноманітність. Якщо на першому поколінні одна особина має пристосованість 100, а всі інші - близько 10, перша отримає приблизно $\frac{100}{100+9\cdot 10} \approx 53\%$ місць у батьківській популяції, і дуже швидко вся популяція стане її копіями. Також метод не працює з від'ємними значеннями пристосованості.

Приклад: Популяція чотирьох особин для задачі максимізації з пристосованостями

$f = (10,20,30,40)$.

Сумарна пристосованість $S = 100$. Ймовірності:

$p = (0.1,0.2,0.3,0.4)$.

Для вибору одного батька генеруємо випадкове число $r \in [0,1]$.

Якщо $r \leq 0.1$, вибираємо першу особину;

якщо $0.1 < r \leq 0.3$, вибираємо другу;

якщо $0.3 < r \leq 0.6$, вибираємо третю;

якщо $r > 0.6$, вибираємо четверту.

Псевдокод пропорційної селекції:

```
функція RouletteWheelSelection(population, fitness, N_select):
    total_fitness = сума(fitness)
    probabilities = [f / total_fitness для f в fitness]
    cumulative = [0]
    для i від 0 до len(probabilities) - 1:
        cumulative.append(cumulative[i] + probabilities[i])

    selected = []
    для _ від 0 до N_select:
        r = випадкове_число(0, 1)
        для i від 0 до len(cumulative) - 1:
            якщо cumulative[i] <= r < cumulative[i+1]:
                selected.append(population[i])
                break

    повернути selected
```

Ранг-селекція

Ранг-селекція усуває проблему масштабу пристосованості. Особини сортуються за пристосованістю, і ймовірність вибору базується не на абсолютному значенні пристосованості, а на рангу особини в відсортованому списку. Найгіршій особині присвоюється ранг 1, наступній - 2, і так далі до найкращої з рангом N. Ймовірність вибору пропорційна рангу:

$$p_i = \frac{rank(i)}{\sum_{j=1}^{N} j} = \frac{2 \cdot rank(i)}{N(N+1)}$$

де використано формулу суми арифметичної прогресії

$\sum_{j=1}^{N} j = \frac{N(N+1)}{2}$.

Це лінійна ранг-селекція. Існує також нелінійна версія, де ймовірність може залежати від рангу нелінійно, наприклад, $p_i \propto rank(i)^{\alpha}$ для деякого показника $\alpha > 1$, що посилює тиск на користь кращих особин.

Переваги ранг-селекції: вона працює з будь-якими значеннями пристосованості, включаючи від’ємні. Селекційний тиск постійний і не залежить від абсолютних значень пристосованостей.

Це запобігає як передчасній збіжності на ранніх етапах, так і застою на пізніх етапах, коли всі пристосованості стають близькими.

Приклад: Та ж популяція з $f = (10,20,30,40)$. Після сортування ранги: особина з $f = 10$ має ранг 1, з $f = 20$ - ранг 2, з $f = 30$ - ранг 3, з $f = 40$ - ранг 4. Ймовірності: $p_1 = \frac{2 \cdot 1}{4 \cdot 5} = 0.1$, $p_2 = 0.2$, $p_3 = 0.3$, $p_4 = 0.4$. Випадково, для цього прикладу ймовірності співпали з пропорційною селекцією, але це тому, що пристосованості вже були пропорційні рангам.

Якщо б пристосованості були (10,20,30,1000), пропорційна селекція дала б (0.01,0.02,0.03,0.94) - майже вся популяція стала б копіями четвертої особини.

Ранг-селекція все одно дала б (0.1,0.2,0.3,0.4) - помірний тиск.

Псевдокод ранг-селекції:

```
функція RankSelection(population, fitness, N_select):
    N = довжина(population)
    # Створюємо пари (індекс, пристосованість) і сортуємо
    indexed_pop = [(i, f) для i, f в enumerate(fitness)]
    indexed_pop.sort(key=lambda x: x[1])  # Сортування за пристосован
істю

    # Присвоюємо ранги
    ranks = [0] * N
    для rank, (i, _) в enumerate(indexed_pop, start=1):
        ranks[i] = rank

    # Обчислюємо ймовірності на основі рангів
    total_ranks = N * (N + 1) / 2
    probabilities = [r / total_ranks для r в ranks]

    # Виконуємо селекцію як roulette wheel
    cumulative = [0]
    для i від 0 до N - 1:
        cumulative.append(cumulative[i] + probabilities[i])

    selected = []
    для _ від 0 до N_select:
        r = випадкове_число(0, 1)
        для i від 0 до N:
            якщо cumulative[i] <= r < cumulative[i+1]:
                selected.append(population[i])
                break
```

```
    повернути selected
```

Турнірна селекція (Tournament Selection)

Турнірна селекція є найпопулярнішим методом в сучасних генетичних алгоритмах завдяки простоті, ефективності та природній паралелізації. Для вибору одного батька випадково вибираються k особин з популяції (з поверненням або без), і найкраща з них за пристосованістю стає батьком. Процес повторюється для кожного батька, що потрібен.

Параметр k називається розміром турніру і контролює селекційний тиск.

При $k = 2$ (бінарний турнір) кожна особина має шанс бути вибраною, якщо вона краща за хоча б одну іншу особину.

При $k = N$ завжди вибирається найкраща особина популяції - максимальний селекційний тиск, що може призвести до передчасної збіжності. Типові значення k лежать в діапазоні від 2 до 7.

Ймовірність, що конкретна особина з рангом r (де 1 - найгірша, N - найкраща) буде вибрана в турнірі розміру k, складна для точного обчислення, але для випадку вибору без повернення і великого N наближено:

$$p_{tournament}(r) \approx \frac{k}{N} \cdot \left(\frac{r}{N}\right)^{k-1}$$

Це показує, що кращі особини (з великим r) мають суттєво вищу ймовірність при великому k.

Турнірна селекція має численні переваги: вона проста у реалізації, не вимагає сортування всієї популяції або обчислення сум, природно паралелізується (кожен турнір незалежний), легко регулюється селекційний тиск через параметр k, працює з від'ємними та нульовими значеннями пристосованості.

Приклад: Популяція з $N = 100$, турнір $k = 3$. Для вибору одного батька генеруємо три випадкові індекси, скажімо {15,67,23}. Дивимося їхні пристосованості:

$f_{15} = 45$, $f_{67} = 78$, $f_{23} = 52$.

Найбільша - $f_{67} = 78$, тому вибираємо 67-у особину як батька.

Псевдокод турнірної селекції:

```
функція TournamentSelection(population, fitness, N_select, k):
    selected = []
```

```
    N = довжина(population)

    для _ від 0 до N_select:
        # Випадково вибираємо k індексів
        tournament_indices = [випадковий_індекс(0, N-1) для _ від 0 д
о k]

        # Знаходимо найкращого в турнірі
        best_index = tournament_indices[0]
        best_fitness = fitness[best_index]

        для idx в tournament_indices[1:]:
            якщо fitness[idx] > best_fitness:
                best_index = idx
                best_fitness = fitness[idx]

        selected.append(population[best_index])

    повернути selected
```

Елітизм

Елітизм - це не окремий метод селекції, а стратегія, що комбінується з будь-яким методом селекції. Ідея проста: n_e найкращих особин поточної популяції автоматично копіюються в наступне покоління без змін, гарантуючи, що найкраще знайдене розв'язок не втрачається через стохастичні оператори.

Без елітизму можлива ситуація, коли найкраща особина не вибирається для розмноження через випадковість селекції, або вибирається, але її нащадки після кросинговеру та мутації гірші за батька. З елітизмом найкраще розв'язок монотонно покращується або залишається незмінним: $\max_i f_i(t+1) \geq \max_i f_i(t)$ для всіх t.

Існує теорема про збіжність:

Теорема про збіжність. Генетичний алгоритм з елітизмом (принаймні один еліт) та достатньо високою ймовірністю мутації збігається до глобального оптимуму з ймовірністю 1 при $t \to \infty$.

Звичайно, це асимптотичний результат, що не гарантує швидкої збіжності на практиці, але він теоретично обґрунтовує ефективність елітизму.

Типово використовують $n_e = 1$ (зберігаємо найкращу особину) або $n_e = 0.01 \cdot N$ до $0.05 \cdot N$ (зберігаємо топ $1-5\%$ популяції). Надмірний елітизм

(великий n_e) може знизити різноманітність і уповільнити дослідження нових областей.

Псевдокод елітизму:

```
функція ApplyElitism(old_population, old_fitness, new_population, n_elite):
    # Знаходимо індекси n_elite найкращих особин
    elite_indices = індекси_n_найкращих(old_fitness, n_elite)

    # Замінюємо перші n_elite особин нової популяції елітами
    для і від 0 до n_elite - 1:
        new_population[i] = old_population[elite_indices[i]]

    повернути new_population
```

Оператори кросинговеру (кросоверу)

Кросовер або схрещування комбінує генетичний матеріал двох батьків для створення нащадків. Це основний оператор дослідження в генетичному алгоритмі. Фундаментальна ідея: якщо два різні розв'язок мають хороші властивості, можливо, їхня комбінація успадкує переваги обох.

Одноточковий кросовер

Для рядкових представлень (бінарних, цілочисельних, символьних). Вибирається випадкова точка розрізу $c \in \{1, \dots, l-1\}$, де l - довжина рядка. Два батьки розрізаються в цій точці, і сегменти обмінюються:

Батько $\mathbf{p}_1 = (p_{1,1}, \dots, p_{1,c}, |p_{1,c+1}, \dots, p_{1,l})$

Батько $\mathbf{p}_2 = (p_{2,1}, \dots, p_{2,c}, |p_{2,c+1}, \dots, p_{2,l})$

Нащадки:

$\mathbf{o}_1 = (p_{1,1}, \dots, p_{1,c}, p_{2,c+1}, \dots, p_{2,l})$

$\mathbf{o}_2 = (p_{2,1}, \dots, p_{2,c}, p_{1,c+1}, \dots, p_{1,l})$

Теорема схем Холланда показує, що ймовірність руйнування схеми H з довжиною визначення $\delta(H)$ при одноточковому кросинговері:

$$p_{disruption} = \frac{\delta(H)}{l-1}$$

Це означає, що компактні схеми (з малою δ) мають більше шансів зберегтися і передатися нащадкам разом. Це забезпечує ефективне збереження та комбінування "будівельних блоків".

Приклад: Задача відбору ознак у машинному навчанні, 10 ознак.

Батько 1: **1100110001** (використовує ознаки 1, 2, 5, 6, 10)

Батько 2: **0011001110** (використовує ознаки 3, 4, 7, 8, 9)

Точка розрізу: $c = 5$

Нащадок 1: **11001** + **01110** = **1100101110** (ознаки 1, 2, 5, 7, 8, 9)

Нащадок 2: **00110** + **10001** = **0011010001** (ознаки 3, 4, 6, 10)

Нащадок 1 об'єднав ознаки {1,2,5} від першого батька з {7,8,9} від другого, створивши нову комбінацію.

Псевдокод одноточкового кросинговеру:

```
функція SinglePointCrossover(parent1, parent2):
    l = довжина(parent1)
    c = випадкове_ціле(1, l-1)  # Точка розрізу

    offspring1 = parent1[0:c] + parent2[c:l]
    offspring2 = parent2[0:c] + parent1[c:l]

    повернути offspring1, offspring2
```

Дв оточковий кросовер

Узагальнення одноточкового. Вибираються дві точки розрізу $c_1 < c_2$, і обмінюється середній сегмент між ними:

Батько $\mathbf{p}_1 = (p_{1,1}, \dots, p_{1,c_1}, |p_{1,c_1+1}, \dots, p_{1,c_2}, |p_{1,c_2+1}, \dots, p_{1,l})$

Батько $\mathbf{p}_2 = (p_{2,1}, \dots, p_{2,c_1}, |p_{2,c_1+1}, \dots, p_{2,c_2}, |p_{2,c_2+1}, \dots, p_{2,l})$

Нащадки:

$\mathbf{o}_1 = (p_{1,1}, \dots, p_{1,c_1}, p_{2,c_1+1}, \dots, p_{2,c_2}, p_{1,c_2+1}, \dots, p_{1,l})$

$\mathbf{o}_2 = (p_{2,1}, \dots, p_{2,c_1}, p_{1,c_1+1}, \dots, p_{1,c_2}, p_{2,c_2+1}, \dots, p_{2,l})$

Двоточковий кросовер зменшує позиційну залежність порівняно з одноточковим. Якщо уявити хромосому як кільце (останній біт сусідній з першим), двоточковий кросовер стає еквівалентним одноточковому на кільці, усуваючи штучну границю між кінцем та початком рядка.

Приклад:

Батько 1: **1110011010**

Батько 2: **0001101101**

Точки: $c_1 = 2, c_2 = 7$

Нащадок 1:

11 + **01101** + **10** = **1101101110**

Нащадок 2:

00 + **10011** + **01** = **0010011001**

Псевдокод двоточкового кросинговеру:

```
функція TwoPointCrossover(parent1, parent2):
    l = довжина(parent1)
    c1 = випадкове_ціле(1, l-2)
    c2 = випадкове_ціле(c1+1, l-1)

    offspring1 = parent1[0:c1] + parent2[c1:c2] + parent1[c2:l]
    offspring2 = parent2[0:c1] + parent1[c1:c2] + parent2[c2:l]

    повернути offspring1, offspring2
```

Рівномірний кросинговер

Замість фіксованих точок розрізу, для кожної позиції незалежно випадково вирішується, від якого батька успадкувати ген. Генерується випадкова бінарна маска $\mathbf{m} = (m_1, \dots, m_l)$, де кожен $m_i \in \{0,1\}$ з ймовірністю 0.5. Для кожної позиції:

$$o_{1,i} = \begin{cases} p_{1,i}, & \text{якщо } m_i = 1 \\ p_{2,i}, & \text{якщо } m_i = 0 \end{cases}$$

$$o_{2,i} = \begin{cases} p_{2,i}, & \text{якщо } m_i = 1 \\ p_{1,i}, & \text{якщо } m_i = 0 \end{cases}$$

Рівномірний кросовер максимізує рекомбінацію - кожен ген нащадка незалежно вибирається від одного з батьків. Це руйнує довгі схеми, але відмінно комбінує короткі незалежні будівельні блоки. Підходить для задач, де гени слабко взаємодіють, наприклад, відбір ознак.

Дисперсія нащадків вище, ніж при одноточковому кросинговері - нащадки можуть бути дуже різними між собою і відрізнятися від обох батьків. Це збільшує дослідження, але може руйнувати хороші комбінації генів, які знаходяться далеко один від одного в рядку.

Приклад:

Батько 1: **1100110001**

Батько 2: **0011001110**

Маска: **1010110100**

Нащадок 1: коли маска=1, беремо від батька 1, коли 0 - від батька 2

Позиції: **1**(б1) **0**(б2) **1**(б1) **0**(б2) **1**(б1) **1**(б1) **0**(б2) **1**(б1) **0**(б2) **0**(б2)

Результат: **1001110010**

Псевдокод рівномірного кросинговеру:

```
функція UniformCrossover(parent1, parent2):
    l = довжина(parent1)
    mask = [випадковий_біт() для _ від 0 до l]

    offspring1 = []
    offspring2 = []

    для i від 0 до l-1:
        якщо mask[i] == 1:
            offspring1.append(parent1[i])
            offspring2.append(parent2[i])
        інакше:
            offspring1.append(parent2[i])
            offspring2.append(parent1[i])

    повернути offspring1, offspring2
```

Арифметичний кросовер

Для дійсного кодування. Нащадки є лінійною комбінацією батьків:

$$\mathbf{o}_1 = \alpha \cdot \mathbf{p}_1 + (1 - \alpha) \cdot \mathbf{p}_2$$
$$\mathbf{o}_2 = (1 - \alpha) \cdot \mathbf{p}_1 + \alpha \cdot \mathbf{p}_2$$

де $\alpha \in [0,1]$ - параметр змішування.

При $\alpha = 0.5$ нащадки є середніми значеннями батьків. Це забезпечує інтермедіацію - нащадки лежать між батьками в просторі параметрів.

Перевага: зберігає допустимість. Якщо батьки задовольняють лінійним обмеженням або обмеженням на діапазони, нащадки також задовольняють їм.

Недолік: обмежене дослідження - нащадки завжди між батьками, неможливо вийти за їхні межі.

Приклад: Настройка гіперпараметрів нейронної мережі

Батько 1: швидкість навчання $\eta_1 = 0.01$, momentum $\mu_1 = 0.9$

Батько 2: $\eta_2 = 0.001$, $\mu_2 = 0.95$ 3 $\alpha = 0.5$:

Нащадок 1: $\eta = 0.5 \cdot 0.01 + 0.5 \cdot 0.001 = 0.0055$, $\mu = 0.5 \cdot 0.9 + 0.5 \cdot 0.95 = 0.925$

Псевдокод арифметичного кросинговеру:

```
функція ArithmeticCrossover(parent1, parent2, alpha=0.5):
    offspring1 = alpha * parent1 + (1 - alpha) * parent2
    offspring2 = (1 - alpha) * parent1 + alpha * parent2

    повернути offspring1, offspring2
```

Кросовер порядку (OX)

Для задач з перестановочним кодуванням (TSP, планування) стандартні оператори не працюють. Кросовер порядку зберігає відносний порядок елементів:

21. Вибираємо два випадкові розрізи c_1, c_2
22. Копіюємо підрядок $[c_1, c_2]$ від першого батька до нащадка
23. Починаючи з позиції $c_2 + 1$, заповнюємо решту позицій нащадка елементами другого батька в порядку їх появи, пропускаючи ті, що вже є в підрядку

Приклад: TSP з 8 містами

Батько 1: `(1 2 3 4 5 6 7 8)`

Батько 2: `(3 5 4 2 1 7 6 8)`

Вибираємо $c_1 = 2$, $c_2 = 5$ (індексація з 0)

Копіюємо підрядок `[2,5]` від батька 1: `(3 4 5 6)`

Нащадок спочатку: `(_ _ | 3 4 5 6 | _ _)`

Елементи батька 2, починаючи з позиції після c_2 і циклічно: `1, 7, 8, 3, 5, 4, 2` Видаляємо вже наявні `(3, 4, 5, 6)`: залишається `1, 7, 8, 2`

Заповнюємо: `(2 1 | 3 4 5 6 | 7 8)`

Псевдокод Кросинговера порядку:

```
функція OrderCrossover(parent1, parent2):
    n = довжина(parent1)
    c1 = випадкове_ціле(0, n-2)
    c2 = випадкове_ціле(c1+1, n-1)

    # Копіюємо середній сегмент
    offspring = [None] * n
    offspring[c1:c2+1] = parent1[c1:c2+1]

    # Список елементів, які вже є в нащадку
    used = set(offspring[c1:c2+1])

    # Заповнюємо решту з parent2, зберігаючи порядок
    idx_offspring = (c2 + 1) % n
```

```
    для elem в parent2[c2+1:] + parent2[:c2+1]:
        якщо elem не в used:
            offspring[idx_offspring] = elem
            idx_offspring = (idx_offspring + 1) % n

    повернути offspring
```

Частково відображений кросовер (PMX)

PMX є іншим популярним оператором для перестановок, який працює інакше ніж OX. Він зберігає абсолютні позиції елементів наскільки можливо. Процедура виглядає так:

24. Вибираємо два випадкові розрізи, визначаючи підрядок для обміну
25. Обмінюємо ці підрядки між батьками
26. Визначаємо відображення між елементами обмінених підрядків
27. Використовуємо це відображення для усунення конфліктів (дублікатів) в решті хромосоми

Ідея полягає в створенні часткового відображення між елементами. Якщо в обмінному сегменті елемент a у першого батька відповідає позиції, де у другого батька стоїть елемент b, то встановлюється відображення $a \leftrightarrow b$. Це відображення використовується для розв'язання конфліктів за межами обмінного сегменту.

Приклад: TSP з 9 містами

Батько 1: (**1 2 3 | 4 5 6 | 7 8 9**)

Батько 2: (**4 5 2 | 1 8 7 | 6 9 3**)

Розрізи на позиціях 3 та 6.

Після обміну середніми сегментами:

Нащадок 1 (спочатку): (**1 2 3 | 1 8 7 | 7 8 9**)

Нащадок 2 (спочатку): (**4 5 2 | 4 5 6 | 6 9 3**)

Бачимо конфлікти - деякі числа повторюються. Будуємо відображення з обмінених сегментів:

Позиція 3: 4→1, позиція 4: 5→8, позиція 5: 6→7

Отже відображення: 4↔1, 5↔8, 6↔7

Тепер виправляємо конфлікти в нащадку 1:

- Позиція 0: елемент 1, він є в середньому сегменті → замінюємо за відображенням 1→4: ставимо 4

- Позиція 1: елемент 2, його немає в середньому сегменті → залишаємо 2

- Позиція 2: елемент 3, його немає в середньому сегменті → залишаємо 3

- Позиції 3-5: середній сегмент, не чіпаємо

- Позиція 6: елемент 7, він є в середньому сегменті → замінюємо за відображенням 7→6: ставимо 6

- Позиція 7: елемент 8, він є в середньому сегменті → замінюємо за відображенням 8→5: ставимо 5

- Позиція 8: елемент 9, його немає → залишаємо 9

Результат: Нащадок 1: (**4 2 3 1 8 7 6 5 9**)

PMX зберігає більше інформації про абсолютні позиції елементів порівняно з OX, що може бути корисним для деяких типів задач планування.

Псевдокод PMX:

```
функція PMX(parent1, parent2):
    n = довжина(parent1)
    c1 = випадкове_ціле(0, n-2)
    c2 = випадкове_ціле(c1+1, n-1)

    offspring = копія(parent1)

    # Обмінюємо середній сегмент
    offspring[c1:c2+1] = parent2[c1:c2+1]

    # Будуємо відображення
    mapping = {}
    для і від c1 до c2:
        mapping[parent2[i]] = parent1[i]
        mapping[parent1[i]] = parent2[i]

    # Виправляємо конфлікти за межами сегменту
    для і в [0:c1] + [c2+1:n]:
        elem = offspring[i]
        # Якщо елемент дублюється (є в середньому сегменті)
        поки elem в offspring[c1:c2+1]:
            elem = mapping[elem]
        offspring[i] = elem

    повернути offspring
```

Кросовер рекомбінації

Кросовер рекомбінації - складніший оператор, спеціально розроблений для TSP. Він намагається зберегти якомога більше ребер (з’єднань між

послідовними містами) з батьківських турів. Ідея полягає в тому, що хороші тури містять хороші ребра (короткі відстані між послідовними містами), і бажано зберегти ці ребра в нащадках.

Алгоритм будує таблицю суміжності, де для кожного міста записуються його сусіди в обох батьківських турах. Потім конструюється нащадок, починаючи з випадкового міста і на кожному кроці вибираючи наступне місто з мінімальною кількістю невідвіданих сусідів. Це евристика прагне використовувати ребра, які мають мало альтернатив.

Приклад спрощеної процедури:

Батько 1: (`1-2-3-4-5-6-1`) - тур

Батько 2: (`1-4-2-5-3-6-1`) - тур

Таблиця ребер (сусідів для кожного міста):

- Місто 1: сусіди {2, 6} від батька 1, {4, 6} від батька 2 → всього {2, 4, 6}

- Місто 2: сусіди {1, 3} від батька 1, {4, 5} від батька 2 → всього {1, 3, 4, 5}

- Місто 3: сусіди {2, 4} від батька 1, {5, 6} від батька 2 → всього {2, 4, 5, 6}

- І так далі…

Побудова нащадка:

1. Починаємо з випадкового міста, наприклад 1

2. Дивимось сусідів міста 1: {2, 4, 6}. Вибираємо того, хто має найменше невідвіданих сусідів

3. Припустимо, обрали

4. Додаємо ребро 1-4 4. Видаляємо 1 з таблиць всіх міст. Поточне місто - 4

5. Сусіди міста 4 (окрім відвіданих): вибираємо наступне

6. Продовжуємо до побудови повного туру

Цей оператор складний в реалізації, але дуже ефективний для TSP, оскільки він інтелектуально комбінує хороші з’єднання з обох батьків.

BLX-α кросовер

Цей кросовер використовується для дійсного кодування. BLX-α розширює ідею арифметичного кросоверу, дозволяючи нащадкам виходити за межі батьківських значень. Для кожної змінної i визначаємо діапазон значень у батьків:

$$c_{min,i} = \min(p_{1,i}, p_{2,i})$$

$$c_{max,i} = \max(p_{1,i}, p_{2,i})$$

$$I_i = c_{max,i} - c_{min,i}$$

Нащадок вибирає значення рівномірно з розширеного діапазону:

$$o_i \in [c_{min,i} - \alpha \cdot I_i, \quad c_{max,i} + \alpha \cdot I_i]$$

Параметр α контролює величину розширення.

Зазвичай, $\alpha = 0.5$.

При $\alpha = 0$ це еквівалентно арифметичному кросоверу з випадковим λ.

При $\alpha > 0$ нащадки можуть виходити за межі батьків, що посилює дослідження.

Приклад: Оптимізація двох параметрів

Батько 1: $(x_1, x_2) = (2.0,5.0)$

Батько 2: $(x_1, x_2) = (4.0,3.0)$ З $\alpha = 0.5$:

Для x_1: $c_{min} = 2.0$, $c_{max} = 4.0$, $I = 2.0$

Діапазон: $[2.0 - 0.5 \cdot 2.0, 4.0 + 0.5 \cdot 2.0] = [1.0,5.0]$

Нащадок отримує x_1 рівномірно з [1.0,5.0]

Для x_2: $c_{min} = 3.0$, $c_{max} = 5.0$, $I = 2.0$

Діапазон: $[3.0 - 0.5 \cdot 2.0, 5.0 + 0.5 \cdot 2.0] = [2.0,6.0]$

Нащадок отримує x_2 рівномірно з [2.0,6.0]

Можливий нащадок: (3.7,4.2) - лежить між батьками

Або: (1.5,5.8) - виходить за межі батьків

Псевдокод BLX-α:

```
функція BLX_alpha(parent1, parent2, alpha=0.5):
    n = довжина(parent1)
    offspring = []

    для і від 0 до n-1:
        c_min = min(parent1[i], parent2[i])
        c_max = max(parent1[i], parent2[i])
        I = c_max - c_min

        # Генеруємо значення з розширеного діапазону
        low = c_min - alpha * I
        high = c_max + alpha * I
        offspring[i] = випадкове_дійсне(low, high)
```

```
    повернути offspring
```

Симульований бінарний кросовер (SBX)

SBX розроблений для імітації поведінки одноточкового кросоверу для дійсних змінних. Він використовує розподіл ймовірностей, що контролюється індексом розподілу η_c. Для кожної змінної генеруються два нащадки:

$$o_{1,i} = 0.5[(1 + \beta_i)p_{1,i} + (1 - \beta_i)p_{2,i}]$$
$$o_{2,i} = 0.5[(1 - \beta_i)p_{1,i} + (1 + \beta_i)p_{2,i}]$$

де параметр β_i визначається з наступного розподілу:

$$\beta_i = \begin{cases} (2u_i)^{1/(\eta_c+1)}, & \text{якщо } u_i \le 0.5 \\ \left(\frac{1}{2(1-u_i)}\right)^{1/(\eta_c+1)}, & \text{якщо } u_i > 0.5 \end{cases}$$

де u_i - випадкове число з рівномірного розподілу [0,1].

Індекс розподілу η_c контролює близькість нащадків до батьків:

- Великий η_c (наприклад, 20): нащадки дуже близькі до батьків, мала дисперсія

- Малий η_c (наприклад, 2): нащадки можуть бути далеко від батьків, велика дисперсія

Ця властивість дозволяє адаптивно балансувати між експлуатацією (пізні покоління, великий η_c) та дослідженням (ранні покоління, малий η_c).

SBX є стандартним оператором у генетичних багатокритеріальних алгоритмах оптимізації та багатьох інших сучасних еволюційних алгоритмах через його хороші теоретичні властивості та емпіричну ефективність.

Псевдокод SBX:

```
функція SBX(parent1, parent2, eta_c=15):
    n = довжина(parent1)
    offspring1 = []
    offspring2 = []

    для і від 0 до n-1:
        u = випадкове_дійсне(0, 1)

        якщо u <= 0.5:
            beta = (2 * u) ** (1.0 / (eta_c + 1))
```

```
        інакше:
            beta = (1.0 / (2 * (1 - u))) ** (1.0 / (eta_c + 1))

        offspring1[i] = 0.5 * ((1 + beta) * parent1[i] + (1 - beta) *
parent2[i])
        offspring2[i] = 0.5 * ((1 - beta) * parent1[i] + (1 + beta) *
parent2[i])

    повернути offspring1, offspring2
```

Оператори мутації

Мутація вносить випадкові зміни в окремих особин, підтримуючи генетичну різноманітність популяції та запобігаючи передчасній збіжності. Якщо кросовер є оператором рекомбінації існуючого генетичного матеріалу, то мутація вводить новий матеріал, якого не було в популяції. Мутація також діє як оператор локального пошуку, створюючи варіації навколо існуючих рішень.

Бітова мутація

Для бінарного кодування. Кожен біт хромосоми інвертується (змінюється з 0 на 1 або з 1 на 0) з невеликою ймовірністю p_m. Це найпростіший та найпоширеніший оператор мутації для бінарних представлень.

Для біта на позиції i:

$$x_{i'} = \begin{cases} 1 - x_i, & \text{з ймовірністю } p_m \\ x_i, & \text{з ймовірністю } 1 - p_m \end{cases}$$

Типове значення $p_m = 1/l$, де l - довжина хромосоми. Це означає, що в середньому мутує один біт на хромосому. Така ймовірність забезпечує баланс:

- достатня різноманітність для дослідження, але не настільки сильна мутація, щоб руйнувати хороші рішення.

З точки зору теорії схем, мутація руйнує схему порядку $o(H)$ з ймовірністю приблизно $o(H) \cdot p_m$. Оскільки p_m мале (наприклад, 0.001 для хромосоми довжини 1000), навіть схеми високого порядку мають хороші шанси вижити.

Приклад: Задача про рюкзак, 10 предметів, $p_m = 0.1$ (для наочності)

Оригінал: **1001101010**

Для кожного біта генеруємо випадкове число $r \in [0,1]$:

- Біт 0: $r = 0.73 > 0.1$ → не мутує → залишається 1

- Біт 1: $r = 0.05 < 0.1 \rightarrow$ мутує → змінюється з 0 на 1

- Біт 2: $r = 0.92 > 0.1 \rightarrow$ не мутує → залишається 0

- Біт 3: $r = 0.84 > 0.1 \rightarrow$ не мутує → залишається 1

- Біт 4: $r = 0.15 > 0.1 \rightarrow$ не мутує → залишається 1

- Біт 5: $r = 0.08 < 0.1 \rightarrow$ мутує → змінюється з 0 на 1

- Біти 6-9: припустимо, не мутують

Результат: **1101111010**

Псевдокод бітової мутації:

```
функція BitFlipMutation(chromosome, p_m):
    l = довжина(chromosome)
    mutated = копія(chromosome)

    для і від 0 до l-1:
        якщо випадкове_дійсне(0, 1) < p_m:
            mutated[i] = 1 - mutated[i]  # Інверсія біта

    повернути mutated
```

Гаусівська мутація

Для дійсного кодування. До кожної змінної додається випадкове значення з нормального (гаусівського) розподілу з нульовим середнім і стандартним відхиленням σ:

$$x_{i'} = x_i + \mathcal{N}(0, \sigma^2)$$

де $\mathcal{N}(0, \sigma^2)$ позначає випадкове число з нормального розподілу з середнім 0 і дисперсією σ^2.

Стандартне відхилення σ контролює величину мутації. Мале σ призводить до малих змін (локальний пошук, fine-tuning), велике σ - до великих стрибків (глобальне дослідження). Типово σ встановлюється як певна частка від діапазону змінної:

$$\sigma = 0.05 \text{ до } 0.2 \cdot (x_{max} - x_{min})$$

Гаусівський розподіл має властивість: малі зміни більш ймовірні, великі зміни рідкісні. Приблизно 68% значень лежать у межах $\pm\sigma$ від середнього, 95% - у межах $\pm 2\sigma$, 99.7% - у межах $\pm 3\sigma$.

Це природна властивість для локального пошуку - ми хочемо переважно досліджувати околицю поточного рішення, але іноді робити більші стрибки.

Після додавання шуму може знадобитися обрізання, щоб значення залишалося в допустимому діапазоні:

$$x_{i'} = \max(x_{min}, \min(x_{max}, x_i + \mathcal{N}(0, \sigma^2)))$$

Приклад: Налаштування ваги нейронної мережі

Поточна вага: $w = 0.5$, діапазон $[-1,1]$, $\sigma = 0.1$

Генеруємо гаусівський шум: $\mathcal{N}(0,0.01) = -0.037$ (приклад значення)

Нова вага: $w' = 0.5 + (-0.037) = 0.463$

Перевіряємо межі: $0.463 \in [-1,1]$ ✓, залишаємо як є

Псевдокод гаусівської мутації:

```
функція GaussianMutation(chromosome, sigma, x_min, x_max):
    n = довжина(chromosome)
    mutated = копія(chromosome)

    для i від 0 до n-1:
        noise = гаусівський_шум(0, sigma)
        mutated[i] = mutated[i] + noise
        # Обрізання до допустимого діапазону
        mutated[i] = max(x_min[i], min(x_max[i], mutated[i]))

    повернути mutated
```

Рівномірна мутація

Альтернативний оператор для дійсного кодування. Змінна замінюється випадковим значенням з рівномірного розподілу на всьому допустимому діапазоні:

$$x_{i'} = x_{i,min} + \text{rand}(0,1) \cdot (x_{i,max} - x_{i,min})$$

де rand(0,1) - рівномірне випадкове число з інтервалу [0,1].

На відміну від гаусівської мутації, яка створює малі зміни навколо поточного значення, рівномірна мутація може перестрибнути в будь-яку точку діапазону з однаковою ймовірністю. Це агресивніший оператор глобального дослідження.

Зазвичай рівномірна мутація застосовується з певною ймовірністю p_m до кожної змінної окремо, а не до всіх змінних одночасно:

Псевдокод рівномірної мутації:

```
функція UniformMutation(chromosome, p_m, x_min, x_max):
    n = довжина(chromosome)
    mutated = копія(chromosome)
```

```
    для і від 0 до n-1:
        якщо випадкове_дійсне(0, 1) < p_m:
            mutated[i] = x_min[i] + випадкове_дійсне(0, 1) * (x_max[i
] - x_min[i])

    повернути mutated
```

Порівняння: Якщо поточне значення $x = 0.3$ в діапазоні [0,1]:

- Гаусівська мутація з $\sigma = 0.05$: швидше за все дасть значення близько $0.25 - 0.35$

- Рівномірна мутація: може дати будь-яке значення від 0 до 1 з однаковою ймовірністю

Полінноміальна мутація

Використовується генетичних багатокритеріальних алгоритмах оптимізації та інших сучасних алгоритмах. Подібна до гаусівської, але використовує поліноміальний розподіл ймовірностей, що дає кращий контроль над формою розподілу:

$$x_{i'} = x_i + (x_{i,max} - x_{i,min}) \cdot \delta_i$$

де δ_i обчислюється з поліноміального розподілу:

$$\delta_i = \begin{cases} (2u_i)^{1/(\eta_m+1)} - 1, & \text{якщо } u_i < 0.5 \\ 1 - (2(1-u_i))^{1/(\eta_m+1)}, & \text{якщо } u_i \geq 0.5 \end{cases}$$

де $u_i \in [0,1]$ - випадкове число, η_m - індекс розподілу мутації (mutation distribution index).

Параметр η_m контролює форму розподілу:

- Великий η_m (наприклад, 20-50): більшість мутацій малі, розподіл зосереджений близько до поточного значення

- Малий η_m (наприклад, 1-5): можливі великі мутації, широкий розподіл

Ця параметризація дозволяє адаптувати силу мутації під час еволюції або для різних типів задач.

Псевдокод полінноміальної мутації:

```
функція PolynomialMutation(chromosome, p_m, eta_m, x_min, x_max):
    n = довжина(chromosome)
    mutated = копія(chromosome)

    для і від 0 до n-1:
        якщо випадкове_дійсне(0, 1) < p_m:
```

```
            u = випадкове_дійсне(0, 1)

            якщо u < 0.5:
                delta = (2 * u) ** (1.0 / (eta_m + 1)) - 1
            інакше:
                delta = 1 - (2 * (1 - u)) ** (1.0 / (eta_m + 1))

            mutated[i] = mutated[i] + (x_max[i] - x_min[i]) * delta
            mutated[i] = max(x_min[i], min(x_max[i], mutated[i]))

    повернути mutated
```

Мутації для перестановок

Для перестановочного кодування потрібні спеціальні оператори, що зберігають властивість перестановки - кожен елемент зустрічається рівно один раз.

Обмінна мутація: Вибираються дві випадкові позиції, і їхні елементи обмінюються місцями. Це найпростіший оператор для перестановок.

$$\pi' = \pi, \quad \text{але } \pi'_i = \pi_j, \quad \pi'_j = \pi_i$$

де i, j - випадково обрані різні позиції.

Приклад: TSP маршрут

Оригінал: `(1 2 3 4 5 6 7 8)`

Випадкові позиції: 2 та 6 (індексація з 0)

Результат: `(1 2 7 4 5 6 3 8)`

Елементи на позиціях 2 і 6 (значення 3 і 7) обмінялися.

Псевдокод Swap Mutation:

```
функція SwapMutation(permutation):
    n = довжина(permutation)
    mutated = копія(permutation)

    i = випадкове_ціле(0, n-1)
    j = випадкове_ціле(0, n-1)
    поки j == i:  # Переконуємося, що позиції різні
        j = випадкове_ціле(0, n-1)

    # Обмін
    mutated[i], mutated[j] = mutated[j], mutated[i]

    повернути mutated
```

Мутація вставкою: Вибирається елемент з однієї позиції і вставляється в іншу позицію, зсуваючи проміжні елементи.

Приклад:

Оригінал: `(1 2 3 4 5 6 7 8)`

Вибираємо позицію 2 (елемент 3), вставляємо на позицію 6

Процес: видаляємо 3 → `(1 2 4 5 6 7 8)`, вставляємо після позиції 5 → `(1 2 4 5 6 3 7 8)`

Псевдокод Insert Mutation:

```
функція InsertMutation(permutation):
    n = довжина(permutation)
    mutated = копія(permutation)

    i = випадкове_ціле(0, n-1)  # Звідки беремо
    j = випадкове_ціле(0, n-1)  # Куди вставляємо

    якщо i == j:
        повернути mutated  # Нічого не змінилося

    elem = mutated[i]
    видалити mutated[i]
    вставити elem на позицію j

    повернути mutated
```

Інверсійна мутація: Вибирається підрядок і переставляється у зворотному порядку. Цей оператор відповідає операції 2-opt у локальному пошуку для TSP.

Приклад:

Оригінал: `(1 2 3 4 5 6 7 8)`

Вибираємо підрядок від позиції 2 до 5: `(3 4 5 6)`

Інвертуємо: `(6 5 4 3)`

Результат: `(1 2 6 5 4 3 7 8)`

Для TSP інверсія підрядка відповідає видаленню двох ребер туру і з'єднанню підмаршруту у зворотному напрямку. Якщо початковий тур мав ребра 2→3 і 6→7, після інверсії будуть ребра 2→6 і 3→7.

Псевдокод Inversion Mutation:

```
функція InversionMutation(permutation):
    n = довжина(permutation)
```

```
    mutated = копія(permutation)

    i = випадкове_ціле(0, n-2)
    j = випадкове_ціле(i+1, n-1)

    # Інвертуємо підрядок [i, j]
    mutated[i:j+1] = реверс(mutated[i:j+1])

    повернути mutated
```

Перемішувальна мутація: Вибирається підрядок і його елементи випадков о перемішуються. Це більш агресивна мутація, ніж swap - змінює відраз у кілька позицій.

Приклад:

Оригінал: `(1 2 3 4 5 6 7 8)`

Вибираємо підрядок від позиції 2 до 5: `(3 4 5 6)`

Перемішуємо: можливий варіант `(5 6 3 4)`

Результат: `(1 2 5 6 3 4 7 8)`

Псевдокод Scramble Mutation:

```
функція ScrambleMutation(permutation):
    n = довжина(permutation)
    mutated = копія(permutation)

    i = випадкове_ціле(0, n-2)
    j = випадкове_ціле(i+1, n-1)

    # Випадково перемішуємо підрядок [i, j]
    substring = mutated[i:j+1]
    перемішати(substring)
    mutated[i:j+1] = substring

    повернути mutated
```

Адаптивні оператори

У базових генетичних алгоритмах параметри операторів (ймовірності кросоверу p_c та мутації p_m, розмір турніру, стандартне відхилення гаусівської мутації) залишаються фіксованими протягом всього процесу еволюції. Проте інтуїтивно зрозуміло, що оптимальні параметри можуть змінюватися з часом.

На ранніх етапах еволюції, коли популяція різноманітна і далека від оптимуму, бажано більше дослідження - сильна мутація, агресивний кросовер.

На пізніх етапах, коли популяція зібралася навколо хороших рішень, потрібна точна настройка - слабка мутація, локальний пошук.

Детерміноване зменшення параметрів

Найпростіший підхід - змінювати параметри за заздалегідь визначеною схемою залежно від номера покоління t. Для мутації часто використовують експоненційне зменшення:

$$\sigma(t) = \sigma_0 \cdot e^{-\lambda t}$$

або лінійне зменшення:

$$\sigma(t) = \sigma_0 \cdot \left(1 - \frac{t}{T}\right)$$

де σ_0 - початкове стандартне відхилення, T - максимальна кількість поколінь, λ - параметр швидкості зменшення.

Додатковий матеріал.

Функція Растрігіна

Функція Растрігіна - це класична тестова функція для алгоритмів глобальної оптимізації, запропонована Леонардом Растрігіним у 1974 році. Вона є багатомодальною функцією з великою кількістю локальних мінімумів.

Загальна формула для n змінних

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$$

де:

- $\mathbf{x} = (x_1, x_2, \dots, x_n)$ - вектор вхідних змінних

- n - розмірність простору (кількість змінних)

- Стандартна область визначення: $x_i \in [-5.12, 5.12]$ для всіх i

Функція має єдиний глобальний мінімум у точці:

$$\mathbf{x}^* = (0,0, \dots ,0)$$

з значенням:

$$f(\mathbf{x}^*) = 0$$

Розглянемо деякі спрощені випадки цієї функції.

Одновимірна функція (n=1)

$$f(x) = 10 + x^2 - 10\cos(2\pi x), \quad x \in [-5.12, 5.12]$$

Характеристики:

- Глобальний мінімум: $x^* = 0$, $f(x^*) = 0$
- Локальні мінімуми приблизно в точках $x = k$, де k - ціле число
- Кількість локальних мінімумів: приблизно 20 в стандартній області

Двовимірна функція (n=2)

$$f(x_1, x_2) = 20 + x_1^2 + x_2^2 - 10\cos(2\pi x_1) - 10\cos(2\pi x_2)$$

де $x_1, x_2 \in [-5.12, 5.12]$

Характеристики:

- Глобальний мінімум: $(x_1^*, x_2^*) = (0,0)$, $f(0,0) = 0$
- Кількість локальних мінімумів: приблизно $10^2 = 100$
- Структура: регулярна сітка локальних мінімумів

Тривимірна функція (n=3)

$$f(x_1, x_2, x_3) = 30 + \sum_{i=1}^{3} [x_i^2 - 10\cos(2\pi x_i)]$$

Характеристики:

- Глобальний мінімум: (0,0,0), $f = 0$
- Кількість локальних мінімумів: приблизно $10^3 = 1000$

Структура функції

Функцію Растрігіна можна розкласти на дві компоненти:

$$f(\mathbf{x}) = \underbrace{10n}_{\text{константа}} + \underbrace{\sum_{i=1}^{n} x_i^2}_{\text{параболічна}} - 10 \underbrace{\sum_{i=1}^{n} \cos(2\pi x_i)}_{\text{хвиляста}}$$

Параболічна компонента

$$f_{parabolic}(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$$

Це проста квадратична функція з мінімумом у нулі. Вона створює загальну “чашоподібну” форму з глобальним мінімумом у початку координат.

Хвиляста (модулююча) компонента

$$f_{wave}(\mathbf{x}) = -10 \sum_{i=1}^{n} \cos(2\pi x_i)$$

Косинусоїдальна компонента створює регулярні коливання з періодом 1, що накладаються на параболічну базу. Це створює численні локальні мінімуми.

- Глобальна структура задається параболічною компонентою - чим далі від нуля, тим більше значення функції
- Локальна структура створюється косинусоїдами - регулярна сітка локальних мінімумів
- Складність зростає експоненційно з розмірністю: для розмірності n приблизно 10^n локальних мінімумів

Наведемо основні властивості цієї функції.

1. Багатомодальність

Функція має велику кількість локальних мінімумів, розподілених регулярно по всій області визначення. Кількість локальних мінімумів приблизно 10^n, де n - розмірність.

2. Сепарабельність

Функція є сепарабельною (розділюваною) - її можна записати як суму функцій від окремих змінних:

$$f(\mathbf{x}) = \sum_{i=1}^{n} g(x_i)$$

де $g(x_i) = 10 + x_i^2 - 10\cos(2\pi x_i)$

Це означає, що оптимізація кожної змінної може проводитися незалежно.

3. Масштабованість

Складність функції легко контролювати через розмірність n. Для тестування алгоритмів зазвичай використовують $n = 2,5,10,20,30$.

4. Симетричність

Функція симетрична відносно всіх координатних осей і початку координат:

$$f(x_1, \dots, x_i, \dots, x_n) = f(x_1, \dots, -x_i, \dots, x_n)$$

5. Гладкість

Функція є неперервною, диференційовною і навіть двічі диференційовною всюди. Це дозволяє використовувати як градієнтні, так і безградієнтні методи оптимізації.

Градієнт функції

Частинні похідні функції Растрігіна:

$$\frac{\partial f}{\partial x_i} = 2x_i + 20\pi\sin(2\pi x_i)$$

Повний градієнт:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} 2x_1 + 20\pi\sin(2\pi x_1) \\ 2x_2 + 20\pi\sin(2\pi x_2) \\ \vdots \\ 2x_n + 20\pi\sin(2\pi x_n) \end{pmatrix}$$

У глобальному мінімумі $\mathbf{x}^* = \mathbf{0}$:

$$\nabla f(\mathbf{0}) = \mathbf{0}$$

Приклади значень функції

Для n=2

(x_1, x_2)	$f(x_1, x_2)$	Коментар
(0,0)	0	Глобальний мінімум
(1,0)	1	Локальний мінімум
(0,1)	1	Локальний мінімум
(1,1)	2	Локальний мінімум
(0.5,0)	15.25	Локальний максимум
(2,2)	8	Локальний мінімум
(5,5)	50	Біля границі області

Ця функція широко використовується для тестування оптимізаційних алгоритмів, тому, що вона має такі властивості

1. Багатомодальність: Велика кількість локальних мінімумів перевіряє здатність алгоритму уникати застрягання
2. Відома структура: Регулярне розташування мінімумів дозволяє аналізувати поведінку алгоритму
3. Масштабованість: Легко тестувати на різних розмірностях від 2D до високовимірних просторів
4. Відомий оптимум: Глобальний мінімум точно відомий, що дозволяє точно виміряти ефективність
5. Складність: Достатньо складна для градієнтних методів (які легко застрягають), але розв’язувана глобальними методами

При тестуванні на функції Растрігіна оцінюють:

1. Якість рішення: Наскільки близько до нуля алгоритм знайшов мінімум
    - Відмінно: $f(\mathrm{x}_{found}) < 0.01$
    - Добре: $f(\mathrm{x}_{found}) < 1$
    - Задовільно: $f(\mathrm{x}_{found}) < 10$
2. Надійність: Відсоток успішних запусків (досягнення близько до глобального мінімуму)
3. Швидкість збіжності: Кількість обчислень функції до досягнення заданої точності
4. Масштабованість: Як змінюється ефективність зі зростанням розмірності

Приклад: Оптимізація функції Растрігіна протягом 1000 поколінь

- Покоління 0: $\sigma = 1.0$ (широкий пошук по всьому простору)

- Покоління 100: $\sigma = 0.9 \cdot e^{-0.003 \cdot 100} = 0.67$ (помірний пошук)

- Покоління 500: $\sigma = 0.9 \cdot e^{-0.003 \cdot 500} = 0.20$ (локальний пошук)

- Покоління 900: $\sigma = 0.9 \cdot e^{-0.003 \cdot 900} = 0.06$ (точний пошук)

Псевдокод детермінованої адаптації:

```
функція AdaptiveSigma(t, T, sigma_0, strategy='linear'):
    якщо strategy == 'linear':
        sigma = sigma_0 * (1 - t / T)
    інакше якщо strategy == 'exponential':
        lambda = 0.003  # Параметр швидкості
        sigma = sigma_0 * exp(-lambda * t)

    повернути max(sigma, sigma_min)  # Мінімальна межа
```

Самоадаптація параметрів на основі різноманітності

Ми будемо змінювати параметри генетичних алгоритмів залежно від різноманітності популяції. Коли популяція стає занадто однорідною (всі особини схожі), підвищуємо ймовірність мутації для відновлення різноманітності. Коли популяція різноманітна, знижуємо мутацію.

Різноманітність можна вимірювати як середню відстань між особинами:

$$D(t) = \frac{1}{N(N-1)} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} d\left(\mathbf{x}_i, \mathbf{x}_j\right)$$

де d - деяка метрика відстані (евклідова, Гемінгова тощо).

Ймовірність мутації адаптується:

$$p_m(t) = p_{m,min} + (p_{m,max} - p_{m,min}) \cdot \left(1 - \frac{D(t)}{D_{max}}\right)$$

де D_{max} - максимальна можлива різноманітність (зазвичай різноманітність початкової популяції).

Коли $D(t)$ велике (висока різноманітність), p_m близьке до $p_{m,min}$. Коли $D(t)$ мале (низька різноманітність), p_m зростає до $p_{m,max}$, посилюючи мутацію для введення нового генетичного матеріалу.

Інші генетичні оператори та техніки

Локальний пошук та меметичні алгоритми

Гібридизація генетичних алгоритмів з локальним пошуком створює меметичні алгоритми, названі так на честь меми - одиниці культурної еволюції, аналога гену в культурній сфері. Ідея проста: після генетичних операцій застосовуємо детерміновані методи локального покращення до деяких або всіх нащадків.

Для неперервних задач це може бути градієнтний спуск, метод Ньютона або квазіньютонівські методи. Для комбінаторних задач - жадібні евристики, локальний пошук k-opt, симуляція відпалу на кілька ітерацій.

Приклад для TSP: 2-opt локальний пошук Після кросоверу та мутації отримано новий тур. Застосовуємо 2-opt:

```
функція TwoOptImprovement(tour, distance_matrix):
    improved = True
    поки improved:
        improved = False
        для i від 0 до n-2:
            для j від i+2 до n:
                # Перевіряємо, чи покращить інверсія сегменту [i+1, j
]
                gain = calculate_gain(tour, i, j, distance_matrix)
                якщо gain > 0:
                    tour = reverse_segment(tour, i+1, j)
                    improved = True
    повернути tour
```

Це значно прискорює збіжність. “Чистий” генетичний алгоритм може знайти тур довжиною 1000 за 1000 поколінь. ГА з 2-opt знаходить тур довжиною 950 за 200 поколінь - і швидше, і краще.

Компроміс: локальний пошук затратний обчислювально. Зазвичай застосовують не до всіх особин, а лише до кількох найкращих кожного покоління, або застосовують спрощену версію (обмежена кількість ітерацій).

Ніші та підтримання різноманітності

У багатомодальній оптимізації (функція з кількома глобальними або локальними максимумами) часто потрібно знайти не один, а кілька оптимумів. Базовий генетичний алгоритм збігається до одного оптимуму через селекційний тиск. Методи нішування дозволяють підтримувати субпопуляції навколо різних оптимумів.

Fitness Sharing: Пристосованість особини зменшується пропорційно кількості сусідів:

$$f_{shared}(i) = \frac{f(i)}{\sum_{j=1}^{N} s\,h(d(i,j))}$$

Функція поділу:

$$sh(d) = \begin{cases} 1 - \left(\frac{d}{\sigma_{share}}\right)^{\alpha}, & \text{якщо } d < \sigma_{share} \\ 0, & \text{інакше} \end{cases}$$

де $d(i,j)$ - відстань між особинами i та j (генотипна або фенотипна), σ_{share} - радіус ніші (параметр, що визначає, наскільки далеко поширюється вплив сусідів), α - параметр форми функції (зазвичай $\alpha = 2$).

Тобто, якщо навколо особини i багато сусідів у радіусі σ_{share}, знаменник великий, і $f_{shared}(i)$ значно менше оригінальної пристосованості $f(i)$. Це знижує селекційний тиск у скупчених регіонах, заохочуючи популяцію розподілятися по різних нішах.

Приклад: Функція з двома максимумами $f(x) = \sin(x) + 0.5\sin(3x)$ на $[0,2\pi]$ має два локальні максимуми приблизно в $x_1 \approx 1.3$ та $x_2 \approx 3.6$.

Без sharing: популяція збігається до одного з максимумів (випадково який).

З sharing ($\sigma_{share} = 0.5$): субпопуляція формується навколо кожного максимуму. Особини біля x_1 мають знижену пристосованість через багатьох

сусідів поблизу. Це робить регіон біля x_2 відносно більш привабливим, і частина популяції мігрує туди, досягаючи балансу.

Обробка обмежень

Багато реальних задач мають складні обмеження. Існує кілька підходів до їх обробки в генетичних алгоритмах.

Штрафні функції: Модифікуємо функцію пристосованості, додаючи штраф за порушення обмежень:

$$f_{penalty}(\mathrm{x}) = f(\mathrm{x}) - P \cdot V(\mathrm{x})$$

де P - штрафний коефіцієнт, $V(\mathrm{x})$ - міра порушення обмежень:

$$V(\mathrm{x}) = \sum_{i=1}^{m} \max(0, g_i(\mathrm{x}))^2 + \sum_{j=1}^{p} |\, h_j(\mathrm{x})|^2$$

Вибір P критичний: занадто малий - обмеження ігноруються, занадто великий - пошук застряє на межі допустимої області.

Динамічні штрафи: P збільшується з поколіннями:

$$P(t) = (C \cdot t)^{\beta}$$

де C та β - параметри. Спочатку дозволяємо недопустимі розв'язок для дослідження, потім поступово підвищуємо штраф.

Оператори відновлення: недопустимі розв'язок модифікуються, щоб стати допустимими. Для задачі про рюкзак, якщо вага перевищує місткість, видаляємо найменш цінні предмети до досягнення допустимості.

Псевдокод оператора відновлення для рюкзака:

```
функція RepairKnapsack(solution, weights, values, capacity):
    total_weight = sum(solution[i] * weights[i] для i від 0 до n)

    поки total_weight > capacity:
        # Знайти предмет з найменшою "ефективністю" (value/weight)
        included = [i для i від 0 до n якщо solution[i] == 1]
        worst = min(included, key=lambda i: values[i] / weights[i])

        # Видалити цей предмет
        solution[worst] = 0
        total_weight -= weights[worst]

    повернути solution
```

Пристосованість у генетичних алгоритмах

Допасованість та пристосованість - це два терміни-синоніми, які використовуються в українській науковій літературі як переклади англійського слова fitness. Обидва терміни є правильними:

- Пристосованість - більш усталений термін, що відповідає біологічному поняттю “здатність до пристосування”
- Допасованість - дослівніший переклад, що підкреслює “міру відповідності” рішення до вимог задачі

Я у тексті використовував переважно термін “пристосованість”, але “допасованість” є цілком коректною альтернативою.

Пристосованість (fitness) - це числова міра якості конкретного рішення (особини) відносно задачі оптимізації, що розв’язується. Це центральне поняття генетичних алгоритмів, яке відіграє ту ж роль, що й природний добір у біологічній еволюції.

У біологічному контексті пристосованість організму - це його здатність виживати та розмножуватися в конкретному середовищі. Чим краще організм пристосований, тим більше нащадків він залишить. В генетичних алгоритмах пристосованість рішення - це його “якість” або “корисність” для розв’язання задачі. Чим вища пристосованість, тим більша ймовірність, що це рішення буде вибране для створення нащадків.

Формально, функція пристосованості (fitness function) - це відображення:

$$f: X \to \mathbb{R}$$

де:

- X - простір можливих рішень (простір пошуку)
- $\mathbb{R}$ - множина дійсних чисел
- $f(\mathbf{x})$ - пристосованість рішення $\mathbf{x}$

Для задачі максимізації: чим більше значення $f(\mathbf{x})$, тим краще рішення $\mathbf{x}$.

Для задачі мінімізації функції $g(\mathbf{x})$ потрібно перетворити її на функцію пристосованості одним зі способів:

$$f(\mathbf{x}) = -g(\mathbf{x})$$

або

$$f(\mathbf{x}) = \frac{1}{1 + g(\mathbf{x})}$$

або

$$f(\mathbf{x}) = C - g(\mathbf{x})$$

де C - достатньо велика константа.

Біологічна аналогія

Розглянемо аналогію детальніше, щоб глибше зрозуміти концепцію:

У природі:

- Середовище ставить “задачу виживання” перед організмами

- Організми з певними характеристиками (швидкість, сила, камуфляж, інтелект) мають різні шанси вижити

- Ті, що виживають і розмножуються, передають свої гени нащадкам

- “Пристосованість” організму неявно виміряється кількістю нащадків, що досягли репродуктивного віку

У генетичному алгоритмі:

- Задача оптимізації ставить “вимоги” до рішень

- Рішення з певними параметрами мають різну якість (точність, швидкість, вартість тощо)

- Ті рішення, що мають вищу пристосованість, частіше вибираються як “батьки” для створення нових рішень

- Пристосованість явно обчислюється функцією $f(\mathbf{x})$

Приклади функцій пристосованості

Приклад 1: Задача про рюкзак

Маємо рюкзак місткістю W і n предметів з вагами w_i та цінностями v_i. Рішення - бінарний вектор $\mathbf{x} = (x_1, \dots, x_n)$, де $x_i = 1$ означає “взяти предмет i”.

Функція пристосованості:

$$f(\mathbf{x}) = \begin{cases} \sum_{i=1}^{n} v_i\, x_i, & \text{якщо } \sum_{i=1}^{n} w_i\, x_i \le W \\ 0, & \text{інакше} \end{cases}$$

Тут пристосованість = сумарна цінність предметів у рюкзаку, якщо не порушено обмеження ваги. Якщо вага перевищена, рішення повністю недопустиме і отримує нульову пристосованість.

Інтерпретація: Чим цінніші предмети вдалося запакувати, не перевищивши вагу, тим “краще пристосоване” це рішення до вимог задачі.

Приклад 2: Задача комівояжера (TSP)

Потрібно знайти найкоротший маршрут через n міст. Рішення - перестановка $\pi = (\pi_1, \dots, \pi_n)$, що визначає порядок відвідування міст.

Довжина маршруту:

$$L(\pi) = \sum_{i=1}^{n-1} d_{\pi_i,\pi_{i+1}} + d_{\pi_n,\pi_1}$$

де d_{ij} - відстань між містами i та j.

Оскільки потрібно мінімізувати довжину, функція пристосованості:

$$f(\pi) = \frac{1}{L(\pi)}$$

або

$$f(\pi) = L_{max} - L(\pi)$$

де L_{max} - верхня оцінка можливої довжини маршруту.

Інтерпретація: Коротший маршрут = вища пристосованість. Рішення, що знаходить ефективний шлях, краще “пристосоване” до мети мінімізації відстані.

Приклад 3: Оптимізація функції

Максимізація функції Растрігіна (для демонстрації):

$$g(\mathbf{x}) = 10n + \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i)\right)$$

Оскільки ця функція зазвичай мінімізується (має мінімум у $\mathbf{x} = \mathbf{0}$ з $g(\mathbf{0}) = 0$), для генетичного алгоритму визначаємо пристосованість:

$$f(\mathbf{x}) = -g(\mathbf{x})$$

Чим ближче $\mathbf{x}$ до оптимуму, тим менше $g(\mathbf{x})$, і тим більше $f(\mathbf{x})$ (вища пристосованість).

Приклад 4: Навчання нейронної мережі

Потрібно знайти оптимальні гіперпараметри мережі. Рішення $\mathbf{x}$ кодує архітектуру та параметри навчання.

Функція пристосованості - точність на валідаційній вибірці:

$$f(\mathbf{x}) = \text{Accuracy}_{validation}(\mathbf{x})$$

або з урахуванням складності моделі:

$$f(\mathbf{x}) = \alpha \cdot \text{Accuracy}_{validation}(\mathbf{x}) - \beta \cdot \frac{N_{parameters}}{10^6}$$

де α та β - вагові коефіцієнти, що балансують точність та компактність.

Інтерпретація: Мережа, що краще класифікує дані і при цьому не є надмірно складною, має вищу пристосованість.

Властивості гарної функції пристосованості

Функція пристосованості повинна задовольняти певним вимогам для ефективної роботи генетичного алгоритму:

1. Обчислюваність

Функція повинна бути обчислюваною за розумний час. Якщо оцінка одного рішення займає години, а алгоритм потребує тисяч оцінок, це стає практично неможливим.

Приклад проблеми: Якщо функція пристосованості вимагає повного фізичного моделювання (CFD, FEA), кожна оцінка може займати години. У такому випадку потрібні спеціальні стратегії (метамоделювання, паралелізація).

2. Монотонність відносно якості

Кращі рішення повинні мати вищу пристосованість. Здається очевидним, але при складному перетворенні цільової функції в функцію пристосованості це може порушитися.

Неправильно:

$$f(\mathbf{x}) = \sin(g(\mathbf{x}))$$

Тут через періодичність синуса відношення порядку може порушитися.

Правильно:

$$f(\mathbf{x}) = -g(\mathbf{x})$$

3. Гладкість (бажано, але не обов’язково)

Невеликі зміни в рішенні повинні призводити до невеликих змін пристосованості. Це допомагає генетичним операторам поступово покращувати рішення.

Приклад негладкості: Функція з різкими стрибками:

$$f(\mathbf{x}) = \begin{cases} 100, & \text{якщо } g(\mathbf{x}) < \epsilon \\ 0, & \text{інакше} \end{cases}$$

Така функція дає дуже мало інформації - рішення або “ідеальне”, або “безнадійне”. Генетичному алгоритму важко орієнтуватися.

Краще: Використовувати безперервну функцію, що поступово зростає при наближенні до оптимуму.

4. Дискримінативність

Функція повинна розрізняти різні за якістю рішення. Якщо всі рішення мають майже однакову пристосованість, селекція стає практично випадковою.

Приклад проблеми: Якщо всі значення пристосованості в діапазоні [100,100.001], різниця настільки мала, що селекційний тиск майже зникає.

Рішення: Масштабування пристосованості, наприклад:

$$f'(\mathbf{x}) = (f(\mathbf{x}) - f_{min})^2$$

5. Відсутність передумов

Функція не повинна робити припущень про структуру рішення, які можуть обмежити пошук. Вона повинна лише оцінювати якість.

Масштабування пристосованості

Іноді необхідно перетворити значення пристосованості для покращення роботи алгоритму:

Лінійне масштабування

$$f'(\mathbf{x}) = a \cdot f(\mathbf{x}) + b$$

де коефіцієнти a та b вибираються так, щоб середня пристосованість популяції дорівнювала певному значенню, а найкраща особина мала задану перевагу.

Степеневе масштабування

$$f'(\mathbf{x}) = f(\mathbf{x})^k$$

де $k > 1$ підсилює різницю між хорошими та поганими рішеннями.

Сигмоїдне масштабування

$$f'(\mathbf{x}) = \frac{1}{1 + e^{-\alpha(f(\mathbf{x}) - \beta)}}$$

де α контролює крутість переходу, β - точку перегину.

Пристосованість та цільова функція

Важливо розуміти різницю:

Цільова функція - це функція, яку ми хочемо оптимізувати в оригінальній задачі. Вона може бути:

- Мінімізується або максимізується

- Мати обмеження

- Бути багатокритеріальною

Функція пристосованості - це перетворена версія цільової функції, яку використовує генетичний алгоритм. Вона завжди:

- Максимізується

- Враховує обмеження (через штрафи або інші механізми)

- Є скалярною (одне число на рішення)

Приклад перетворення:

Оригінальна задача:

$$\min g(\mathbf{x}) \text{ при } h(\mathbf{x}) \leq 0$$

Функція пристосованості:

$$f(\mathbf{x}) = -g(\mathbf{x}) - P \cdot \max(0, h(\mathbf{x}))^2$$

де P - штрафний коефіцієнт за порушення обмеження.

Роль пристосованості в генетичному алгоритмі

Пристосованість визначає еволюційну динаміку популяції:

1. Селекція: Особини з вищою пристосованістю мають більшу ймовірність бути вибраними як батьки
2. Репродуктивний успіх: Очікувана кількість копій особини i у наступному поколінні пропорційна f_i/f_{avg}
3. Напрямок пошуку: Популяція зміщується в регіони простору з вищою середньою пристосованістю
4. Збіжність: Алгоритм збігається, коли популяція концентрується навколо особин з високою пристосованістю

Уявімо простір пошуку як географічний ландшафт, де:

- Висота в кожній точці = пристосованість рішення в цій точці

- Вершини = хороші рішення (висока пристосованість)

- Долини = погані рішення (низька пристосованість)

- Глобальний максимум = найвища вершина = оптимальне рішення

Генетичний алгоритм - це популяція “дослідників”, що блукають цим ландшафтом:

- Дослідники на вершинах (висока пристосованість) мають більше “нащадків”, які продовжують дослідження поблизу

- Дослідники в долинах (низька пристосованість) мають мало шансів залишити нащадків - З часом популяція концентрується на найвищих вершинах

Пристосованість говорить алгоритму: “Ось тут добре, шукай поблизу”, а в інших місцях: “Тут погано, не витрачай зусилля”.

Наведемо деякі практичні поради щодо проектування функції пристосованості

1. Починайте з простого: Спочатку використайте пряме перетворення цільової функції, потім ускладнюйте при необхідності
2. Тестуйте на відомих рішеннях: Перевірте, що функція дійсно присвоює вищі значення кращим рішенням
3. Нормалізуйте діапазон: Якщо функція може приймати дуже різні значення, нормалізуйте до [0, 1] або [-1, 1]
4. Балансуйте штрафи: Якщо використовуєте штрафні функції, підбирайте коефіцієнти так, щоб штрафи були відчутними, але не домінували
5. Враховуйте обчислювальну вартість: Якщо оцінка дорога, розгляньте спрощені або апроксимовані версії функції

Загальний алгоритм генетичного пошуку

Розглянувши детально оператори, можна переходити до глибокого вивчення генетичного алгоритму.

Генетичний алгоритм є ітераційною стохастичною процедурою пошуку оптимального або близького до оптимального розв'язок в просторі можливих варіантів.

Нехай задано задачу оптимізації: знайти $\mathbf{x}^* \in X$, що максимізує функцію пристосованості $f: X \to \mathbb{R}$, де X - допустима область пошуку.

Генетичний алгоритм оперує не з одним рішенням, а з популяцією розв'язків $P(t) = \{\mathbf{x}_1(t), \mathbf{x}_2(t), \dots, \mathbf{x}_N(t)\}$ в поколінні t, де N - розмір популяції. Кожне розв'язок $\mathbf{x}_i$ називається особиною або індивідом і має відповідне значення пристосованості $f_i = f(\mathbf{x}_i)$.

Структура базового генетичного алгоритму

Загальна структура генетичного алгоритму складається з послідовності кроків, які повторюються циклічно до виконання критерію зупинки:

Крок 1. Ініціалізація Генерується початкова популяція $P(0)$ з N випадкових або евристично сконструйованих рішень. Встановлюється лічильник поколінь $t = 0$. Для кожної особини $\mathrm{x}_i \in P(0)$ обчислюється пристосованість $f_i = f(\mathrm{x}_i)$.

Крок 2. Перевірка критерію зупинки Якщо виконано умову завершення (досягнуто максимальну кількість поколінь T, знайдено розв'язок достатньої якості, вичерпано обчислювальний бюджет або відбулася стагнація), алгоритм зупиняється і повертає найкраще знайдене рішення:

$$\mathbf{x}^* = \arg \max_{\substack{t \\ \mathbf{x}_i \in \bigcup_{s=0} P(s)}} f(\mathbf{x}_i)$$

Інакше переходимо до кроку 3.

Крок 3. Селекція На основі значень пристосованості $\{f_1, f_2, \dots, f_N\}$ вибираються батьківські особини для створення нащадків. Формується проміжна популяція $P_{parent}(t)$ розміру N або $N/2$ (залежно від стратегії) шляхом багаторазового застосування оператора селекції. Ймовірність вибору особини x_i як батька залежить від її відносної пристосованості:

$$p_i = S(f_i, f_1, \dots, f_N)$$

де S - функція селекції (пропорційна, рангова, турнірна тощо).

Крок 4. Кросовер До батьківських особин застосовується оператор кросоверу. Батьки групуються в пари (випадково або послідовно). Для кожної пари $(\mathbf{x}_p, \mathbf{x}_q)$ з ймовірністю p_c (ймовірність кросоверу, типово 0.6-0.9) застосовується оператор кросоверу:

$$(\mathbf{x}'_1, \mathbf{x}'_2) = \text{Crossover}(\mathbf{x}_p, \mathbf{x}_q)$$

що створює двох нащадків $\mathbf{x}'_1$ та $\mathbf{x}'_2$. З ймовірністю $1 - p_c$ нащадки є точними копіями батьків. Всі нащадки збираються в проміжну популяцію $P_{offspring}(t)$.

Крок 5. Мутація До кожного нащадка $x'_i \in P_{offspring}(t)$ застосовується оператор мутації:

$$x''_i = \text{Mutation}(x'_i)$$

Для кожного гену або компоненти розв'язок з ймовірністю p_m (ймовірність мутації, типово 0.001-0.1) застосовується випадкова зміна. Результат зберігається в тій же популяції нащадків, що тепер містить мутовані особини.

Крок 6. Оцінювання нащадків Для кожного нового нащадка $x''_i \in P_{offspring}(t)$ обчислюється його пристосованість $f''_i = f(x''_i)$.

Крок 7. Формування нової популяції (заміна) На основі поточної популяції $P(t)$ та нащадків $P_{offspring}(t)$ формується нова популяція наступного покоління $P(t+1)$ розміру N. Існують різні стратегії заміни:

- Повна заміна: $P(t+1) = P_{offspring}(t)$
- Елітизм: n_e найкращих особин з $P(t)$ гарантовано переходять в $P(t+1)$, решта місць заповнюються нащадками
- Steady-state: замінюються лише кілька найгірших особин з $P(t)$ кращими нащадками

Крок 8. Інкремент лічильника Збільшуємо лічильник покоління: $t = t + 1$. Повертаємося до кроку 2.

Псевдокод базового генетичного алгоритму

```
АЛГОРИТМ: Генетичний_Алгоритм
ВХІД:
    N - розмір популяції
    T - максимальна кількість поколінь
    p_c - ймовірність кросоверу
    p_m - ймовірність мутації
    n_e - кількість елітних особин
    f - функція пристосованості
ВИХІД:
    x_best - найкраще знайдене рішення
    f_best - його пристосованість

ПОЧАТОК
    // Крок 1: Ініціалізація
```

```
t ← 0
P(0) ← ІНІЦІАЛІЗУВАТИ_ПОПУЛЯЦІЮ(N)

ДЛЯ кожної особини x_i в P(0):
    f_i ← ОБЧИСЛИТИ_ПРИСТОСОВАНІСТЬ(x_i)
КІНЕЦЬ_ДЛЯ

x_best ← ЗНАЙТИ_НАЙКРАЩУ_ОСОБИНУ(P(0))
f_best ← f(x_best)

// Основний цикл еволюції
ПОКИ t < T АБО НЕ ВИКОНАНО_КРИТЕРІЙ_ЗУПИНКИ():

    // Крок 2: Селекція батьків
    P_parent ← ПОРОЖНЯ_ПОПУЛЯЦІЯ()
    ДЛЯ i від 1 до N:
        батько ← СЕЛЕКЦІЯ(P(t))
        ДОДАТИ батько ДО P_parent
    КІНЕЦЬ_ДЛЯ

    // Крок 3: Кросовер
    P_offspring ← ПОРОЖНЯ_ПОПУЛЯЦІЯ()
    ДЛЯ i від 1 до N/2:
        // Вибрати двох батьків
        батько1 ← P_parent[2*i - 1]
        батько2 ← P_parent[2*i]

        // Застосувати кросовер з ймовірністю p_c
        ЯКЩО ВИПАДКОВЕ_ЧИСЛО(0,1) < p_c:
            (нащадок1, нащадок2) ← КРОСОВЕР(батько1, батько2)
        ІНАКШЕ:
            нащадок1 ← КОПІЯ(батько1)
            нащадок2 ← КОПІЯ(батько2)
        КІНЕЦЬ_ЯКЩО

        ДОДАТИ нащадок1 ДО P_offspring
        ДОДАТИ нащадок2 ДО P_offspring
    КІНЕЦЬ_ДЛЯ

    // Крок 4: Мутація
    ДЛЯ кожного нащадка в P_offspring:
        нащадок ← МУТАЦІЯ(нащадок, p_m)
    КІНЕЦЬ_ДЛЯ

    // Крок 5: Оцінювання нащадків
```

```
        ДЛЯ кожного нащадка в P_offspring:
            f_нащадка ← ОБЧИСЛИТИ_ПРИСТОСОВАНІСТЬ(нащадок)
        КІНЕЦЬ_ДЛЯ

        // Крок 6: Формування нової популяції (елітизм)
        P(t+1) ← ПОРОЖНЯ_ПОПУЛЯЦІЯ()

        // Додати елітні особини
        elite ← ВИБРАТИ_N_НАЙКРАЩИХ(P(t), n_e)
        ДОДАТИ elite ДО P(t+1)

        // Додати решту нащадків
        решта ← ВИБРАТИ_ВИПАДКОВО(P_offspring, N - n_e)
        ДОДАТИ решта ДО P(t+1)

        // Оновити найкраще рішення
        x_current_best ← ЗНАЙТИ_НАЙКРАЩУ_ОСОБИНУ(P(t+1))
        ЯКЩО f(x_current_best) > f_best:
            x_best ← x_current_best
            f_best ← f(x_current_best)
        КІНЕЦЬ_ЯКЩО

        // Крок 7: Інкремент покоління
        t ← t + 1

        // Виведення прогресу (опціонально)
        ВИВЕСТИ("Покоління", t, ": Найкраща пристосованість =", f_bes
t)

    КІНЕЦЬ_ПОКИ

    ПОВЕРНУТИ x_best, f_best
КІНЕЦЬ
```

Допоміжні процедури:

Ініціалізація популяції

```
ПРОЦЕДУРА: ІНІЦІАЛІЗУВАТИ_ПОПУЛЯЦІЮ(N)
ВХІД: N - розмір популяції
ВИХІД: P - початкова популяція

ПОЧАТОК
    P ← ПОРОЖНІЙ_СПИСОК()

    ДЛЯ і від 1 до N:
        // Генерація випадкової особини
```

```
        x ← СТВОРИТИ_ВИПАДКОВУ_ОСОБИНУ()

        // Перевірка допустимості (якщо є обмеження)
        ПОКИ НЕ ДОПУСТИМА(x):
            x ← СТВОРИТИ_ВИПАДКОВУ_ОСОБИНУ()
        КІНЕЦЬ_ПОКИ

        ДОДАТИ x ДО P
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ P
КІНЕЦЬ
```

Для бінарного кодування:

```
ФУНКЦІЯ: СТВОРИТИ_ВИПАДКОВУ_ОСОБИНУ()
ВИХІД: x - випадкова бінарна хромосома довжини l

ПОЧАТОК
    x ← ПОРОЖНІЙ_ВЕКТОР(l)

    ДЛЯ i від 1 до l:
        x[i] ← ВИПАДКОВИЙ_БІТ()  // 0 або 1 з ймовірністю 0.5
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ x
КІНЕЦЬ
```

Для дійсного кодування:

```
ФУНКЦІЯ: СТВОРИТИ_ВИПАДКОВУ_ОСОБИНУ()
ВИХІД: x - випадковий дійсний вектор розмірності n

ПОЧАТОК
    x ← ПОРОЖНІЙ_ВЕКТОР(n)

    ДЛЯ i від 1 до n:
        // Рівномірний розподіл на [x_min[i], x_max[i]]
        x[i] ← x_min[i] + ВИПАДКОВЕ_ЧИСЛО(0,1) * (x_max[i] - x_min[i]
)
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ x
КІНЕЦЬ
```

Процедури селекції:

Турнірна селекція:

```
ФУНКЦІЯ: СЕЛЕКЦІЯ_ТУРНІРНА(P, k)
ВХІД:
    P - поточна популяція
    k - розмір турніру
ВИХІД: x_winner - переможець турніру

ПОЧАТОК
    турнір ← ПОРОЖНІЙ_СПИСОК()

    // Випадково вибрати k особин
    ДЛЯ і від 1 до k:
        індекс ← ВИПАДКОВИЙ_ІНДЕКС(0, РОЗМІР(P)-1)
        ДОДАТИ P[індекс] ДО турнір
    КІНЕЦЬ_ДЛЯ

    // Знайти найкращу в турнірі
    x_winner ← турнір[0]
    f_winner ← f(x_winner)

    ДЛЯ кожної особини x в турнір:
        ЯКЩО f(x) > f_winner:
            x_winner ← x
            f_winner ← f(x)
        КІНЕЦЬ_ЯКЩО
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ x_winner
КІНЕЦЬ
```

Рулетна селекція:

```
ФУНКЦІЯ: СЕЛЕКЦІЯ_РУЛЕТНА(P)
ВХІД: P - поточна популяція з пристосованостями
ВИХІД: x_selected - вибрана особина

ПОЧАТОК
    N ← РОЗМІР(P)

    // Обчислити сумарну пристосованість
    total_fitness ← 0
    ДЛЯ кожної особини x в P:
        total_fitness ← total_fitness + f(x)
    КІНЕЦЬ_ДЛЯ

    // Обчислити кумулятивні ймовірності
    cumulative ← ПОРОЖНІЙ_МАСИВ(N+1)
```

```
    cumulative[0] ← 0

    ДЛЯ i від 1 до N:
        cumulative[i] ← cumulative[i-1] + f(P[i-1]) / total_fitness
    КІНЕЦЬ_ДЛЯ

    // Крутити рулетку
    r ← ВИПАДКОВЕ_ЧИСЛО(0, 1)

    ДЛЯ i від 1 до N:
        ЯКЩО cumulative[i-1] <= r < cumulative[i]:
            ПОВЕРНУТИ P[i-1]
        КІНЕЦЬ_ЯКЩО
    КІНЕЦЬ_ДЛЯ

    // На всяк випадок повертаємо останню
    ПОВЕРНУТИ P[N-1]
КІНЕЦЬ
```

Процедури кросоверу:

Одноточковий кросовер для бінарного кодування:

```
ФУНКЦІЯ: КРОСОВЕР_ОДНОТОЧКОВИЙ(батько1, батько2)
ВХІД:
    батько1, батько2 - батьківські хромосоми довжини l
ВИХІД:
    нащадок1, нащадок2 - дочірні хромосоми

ПОЧАТОК
    l ← ДОВЖИНА(батько1)

    // Випадково обрати точку розрізу
    точка_розрізу ← ВИПАДКОВЕ_ЦІЛЕ(1, l-1)

    // Створити нащадків
    нащадок1 ← ПОРОЖНІЙ_ВЕКТОР(l)
    нащадок2 ← ПОРОЖНІЙ_ВЕКТОР(l)

    // Копіювати перші частини
    ДЛЯ i від 0 до точка_розрізу - 1:
        нащадок1[i] ← батько1[i]
        нащадок2[i] ← батько2[i]
    КІНЕЦЬ_ДЛЯ

    // Обміняти другі частини
    ДЛЯ i від точка_розрізу до l - 1:
```

```
        нащадок1[i] ← батько2[i]
        нащадок2[i] ← батько1[i]
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ нащадок1, нащадок2
КІНЕЦЬ
```

Арифметичний кросовер для дійсного кодування:

```
ФУНКЦІЯ: КРОСОВЕР_АРИФМЕТИЧНИЙ(батько1, батько2, alpha)
ВХІД:
    батько1, батько2 - батьківські вектори розмірності n
    alpha - коефіцієнт змішування (типово 0.5)
ВИХІД:
    нащадок1, нащадок2 - дочірні вектори

ПОЧАТОК
    n ← РОЗМІРНІСТЬ(батько1)

    нащадок1 ← ПОРОЖНІЙ_ВЕКТОР(n)
    нащадок2 ← ПОРОЖНІЙ_ВЕКТОР(n)

    ДЛЯ i від 0 до n - 1:
        нащадок1[i] ← alpha * батько1[i] + (1 - alpha) * батько2[i]
        нащадок2[i] ← (1 - alpha) * батько1[i] + alpha * батько2[i]
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ нащадок1, нащадок2
КІНЕЦЬ
```

SBX кросовер для дійсного кодування:

```
ФУНКЦІЯ: КРОСОВЕР_SBX(батько1, батько2, eta_c)
ВХІД:
    батько1, батько2 - батьківські вектори
    eta_c - індекс розподілу (типово 15-20)
ВИХІД:
    нащадок1, нащадок2 - дочірні вектори

ПОЧАТОК
    n ← РОЗМІРНІСТЬ(батько1)

    нащадок1 ← ПОРОЖНІЙ_ВЕКТОР(n)
    нащадок2 ← ПОРОЖНІЙ_ВЕКТОР(n)

    ДЛЯ i від 0 до n - 1:
        u ← ВИПАДКОВЕ_ЧИСЛО(0, 1)
```

```
        // Обчислити beta
        ЯКЩО u <= 0.5:
            beta ← (2 * u) ^ (1.0 / (eta_c + 1))
        ІНАКШЕ:
            beta ← (1.0 / (2 * (1 - u))) ^ (1.0 / (eta_c + 1))
        КІНЕЦЬ_ЯКЩО

        // Створити нащадків
        нащадок1[i] ← 0.5 * ((1 + beta) * батько1[i] + (1 - beta) * б
атько2[i])
        нащадок2[i] ← 0.5 * ((1 - beta) * батько1[i] + (1 + beta) * б
атько2[i])

        // Обрізання до допустимих меж
        нащадок1[i] ← ОБРІЗАТИ(нащадок1[i], x_min[i], x_max[i])
        нащадок2[i] ← ОБРІЗАТИ(нащадок2[i], x_min[i], x_max[i])
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ нащадок1, нащадок2
КІНЕЦЬ
```

Процедури мутації:

Бітова мутація для бінарного кодування:

```
ФУНКЦІЯ: МУТАЦІЯ_БІТОВА(особина, p_m)
ВХІД:
    особина - бінарна хромосома довжини l
    p_m - ймовірність мутації одного біта
ВИХІД:
    мутована_особина - мутована хромосома

ПОЧАТОК
    l ← ДОВЖИНА(особина)
    мутована_особина ← КОПІЯ(особина)

    ДЛЯ i від 0 до l - 1:
        ЯКЩО ВИПАДКОВЕ_ЧИСЛО(0, 1) < p_m:
            // Інвертувати біт
            мутована_особина[i] ← 1 - мутована_особина[i]
        КІНЕЦЬ_ЯКЩО
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ мутована_особина
КІНЕЦЬ
```

Гаусівська мутація для дійсного кодування:

```
ФУНКЦІЯ: МУТАЦІЯ_ГАУСІВСЬКА(особина, sigma)
ВХІД:
    особина - дійсний вектор розмірності n
    sigma - стандартне відхилення шуму
ВИХІД:
    мутована_особина - мутований вектор

ПОЧАТОК
    n ← РОЗМІРНІСТЬ(особина)
    мутована_особина ← КОПІЯ(особина)

    ДЛЯ і від 0 до n - 1:
        // Додати гаусівський шум
        шум ← ГАУСІВСЬКИЙ_ШУМ(0, sigma)
        мутована_особина[i] ← мутована_особина[i] + шум

        // Обрізати до допустимих меж
        мутована_особина[i] ← ОБРІЗАТИ(мутована_особина[i], x_min[i],
x_max[i])
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ мутована_особина
КІНЕЦЬ
```

Полінноміальна мутація:

```
ФУНКЦІЯ: МУТАЦІЯ_ПОЛІННОМІАЛЬНА(особина, p_m, eta_m)
ВХІД:
    особина - дійсний вектор
    p_m - ймовірність мутації змінної
    eta_m - індекс розподілу мутації
ВИХІД:
    мутована_особина - мутований вектор

ПОЧАТОК
    n ← РОЗМІРНІСТЬ(особина)
    мутована_особина ← КОПІЯ(особина)

    ДЛЯ і від 0 до n - 1:
        ЯКЩО ВИПАДКОВЕ_ЧИСЛО(0, 1) < p_m:
            u ← ВИПАДКОВЕ_ЧИСЛО(0, 1)

            // Обчислити delta
            ЯКЩО u < 0.5:
                delta ← (2 * u) ^ (1.0 / (eta_m + 1)) - 1
            ІНАКШЕ:
```

```
                delta ← 1 - (2 * (1 - u)) ^ (1.0 / (eta_m + 1))
            КІНЕЦЬ_ЯКЩО

            // Застосувати мутацію
            діапазон ← x_max[i] - x_min[i]
            мутована_особина[i] ← мутована_особина[i] + delta * діапа
зон

            // Обрізати
            мутована_особина[i] ← ОБРІЗАТИ(мутована_особина[i], x_min
[i], x_max[i])
        КІНЕЦЬ_ЯКЩО
    КІНЕЦЬ_ДЛЯ

    ПОВЕРНУТИ мутована_особина
КІНЕЦЬ
```

Мутація перестановок:

```
ФУНКЦІЯ: МУТАЦІЯ_SWAP(перестановка)
ВХІД: перестановка - вектор чисел від 1 до n
ВИХІД: мутована_перестановка

ПОЧАТОК
    n ← ДОВЖИНА(перестановка)
    мутована_перестановка ← КОПІЯ(перестановка)

    // Вибрати дві випадкові позиції
    i ← ВИПАДКОВЕ_ЦІЛЕ(0, n-1)
    j ← ВИПАДКОВЕ_ЦІЛЕ(0, n-1)

    ПОКИ j == i:
        j ← ВИПАДКОВЕ_ЦІЛЕ(0, n-1)
    КІНЕЦЬ_ПОКИ

    // Обміняти елементи
    temp ← мутована_перестановка[i]
    мутована_перестановка[i] ← мутована_перестановка[j]
    мутована_перестановка[j] ← temp

    ПОВЕРНУТИ мутована_перестановка
КІНЕЦЬ
```

Критерії зупинки:

```
ФУНКЦІЯ: ВИКОНАНО_КРИТЕРІЙ_ЗУПИНКИ()
ВИХІД: TRUE якщо потрібно зупинитися, FALSE інакше
```

```
ПОЧАТОК
    // Критерій 1: Досягнуто максимальну кількість поколінь
    ЯКЩО t >= T:
        ПОВЕРНУТИ TRUE
    КІНЕЦЬ_ЯКЩО

    // Критерій 2: Досягнуто цільове значення пристосованості
    ЯКЩО f_best >= f_target:
        ПОВЕРНУТИ TRUE
    КІНЕЦЬ_ЯКЩО

    // Критерій 3: Стагнація (немає покращення протягом k поколінь)
    ЯКЩО t >= k:
        покращення ← |f_best(t) - f_best(t-k)|
        ЯКЩО покращення < epsilon:
            ПОВЕРНУТИ TRUE
        КІНЕЦЬ_ЯКЩО
    КІНЕЦЬ_ЯКЩО

    // Критерій 4: Втрата різноманітності популяції
    різноманітність ← ОБЧИСЛИТИ_РІЗНОМАНІТНІСТЬ(P(t))
    ЯКЩО різноманітність < поріг_різноманітності:
        ПОВЕРНУТИ TRUE
    КІНЕЦЬ_ЯКЩО

    // Критерій 5: Вичерпано обчислювальний бюджет
    ЯКЩО кількість_обчислень_функції >= максимум_обчислень:
        ПОВЕРНУТИ TRUE
    КІНЕЦЬ_ЯКЩО

    ПОВЕРНИТИ FALSE
КІНЕЦЬ
```

Приклад роботи алгоритму

Розглянемо конкретний приклад роботи генетичного алгоритму для простої задачі максимізації функції однієї змінної:

$$f(x) = x \cdot \sin(10\pi x) + 1, \quad x \in [0,1]$$

Використаємо бінарне кодування з кодом Грея.

Параметри:

- Розмір популяції: $N = 6$

- Довжина хромосоми: $l = 10$ біт

- Ймовірність кросоверу: $p_c = 0.8$

- Ймовірність мутації: $p_m = 0.1$

- Кількість поколінь: $T = 20$

Покоління 0 (Ініціалізація):

№	Хромосома (код Грея)	Десяткове	x	$f(x)$
1	0110101010	362	0.353	1.269
2	1101001101	741	0.724	0.541
3	0010110111	189	0.184	0.634
4	1001110001	673	0.657	1.623
5	0111010110	378	0.369	1.234
6	1010011100	540	0.527	0.892

Найкраща особина: №4 з $f(x) = 1.623$

Селекція (турнірна з $k = 2$):

Проводимо шість турнірів для вибору батьків:

- Турнір 1: особини 2 vs 5 → переможець 5 - Турнір 2: особини 1 vs 4 → переможець 4

- Турнір 3: особини 3 vs 6 → переможець 6

- Турнір 4: особини 1 vs 2 → переможець 1

- Турнір 5: особини 4 vs 5 → переможець 4

- Турнір 6: особини 3 vs 6 → переможець 6

Батьківська популяція: [5, 4, 6, 1, 4, 6]

Кросовер:

Формуємо три пари і застосовуємо одноточковий кросовер:

Пара 1: батьки 5 та 4

- Батько 5: **0111010110**

- Батько 4: **1001110001**

- Точка розрізу: 4

- Нащадок 1: **0111** + **110001** = **0111110001**

- Нащадок 2: **1001** + **010110** = **1001010110**

Пара 2: батьки 6 та 1 (припустимо, кросовер не відбувся)

- Нащадок 3: **1010011100** (копія батька 6)

- Нащадок 4: **0110101010** (копія батька 1)

Пара 3: батьки 4 та 6

- Батько 4: **1001110001**

- Батько 6: **1010011100**

- Точка розрізу: 6

- Нащадок 5: **100111** + **1100** = **1001111100**

- Нащадок 6: **101001** + **0001** = **1010010001**

Мутація:

Застосовуємо бітову мутацію до кожного нащадка з $p_m = 0.1$:

Припустимо, мутували наступні біти:

- Нащадок 1: позиція 7 → **0111110001** → **0111110011**

- Нащадки 2-6: без змін (з ймовірністю)

Оцінювання нащадків:

№	Хромосома	Десяткове	x	$f(x)$
1	0111110011	411	0.401	1.305
2	1001010110	630	0.615	1.894
3	1010011100	540	0.527	0.892
4	0110101010	362	0.353	1.269
5	1001111100	684	0.668	1.512
6	1010010001	529	0.516	0.965

Покоління 1:

Нова популяція складається з цих шести нащадків. Найкраща особина тепер: №2 з $f(x) = 1.894$. Бачимо, що отримали покращення.

Після 20 поколінь алгоритм знаходить розв'язок близьке до глобального максимуму, який знаходиться приблизно в точці $x \approx 0.85$ з $f(x) \approx 1.85$.

Математичний аналіз збіжності

Можна показати, що генетичний алгоритм з елітизмом та достатньою ймовірністю мутації збігається до глобального оптимуму з ймовірністю 1:

$$\lim_{t\to\infty} P\left(\max_{\mathbf{x}\in P(t)} f(\mathbf{x}) = f^*\right) = 1$$

де $f^* = \max_{\mathbf{x}\in X} f(\mathbf{x})$ - глобальний оптимум.

Проте швидкість збіжності не гарантована і може бути дуже повільною для складних задач. На практиці генетичні алгоритми зупиняють після

фіксованої кількості поколінь, отримуючи субоптимальне, але практично прийнятне рішення.

Налаштування параметрів

Ефективність генетичного алгоритму критично залежить від вибору параметрів:

Розмір популяції N:

- Малий ($N < 30$): швидка збіжність, але ризик застрягання в локальних оптимумах

- Середній ($N = 50$-200): хороший баланс для більшості задач

- Великий ($N > 500$): повільна збіжність, але кращедослідження

Ймовірність кросоверу p_c:

- Типово 0.6-0.9

- Висока $p_c \rightarrow$ більше дослідження

- Низька $p_c \rightarrow$ більше схоже на випадковий пошук

Ймовірність мутації p_m:

- Для бінарного кодування: $p_m = 1/l$ (один біт на хромосому)

- Для дійсного кодування: $p_m = 1/n$ (одна змінна)

- Дуже важливо для підтримання різноманітності

Селекційний тиск:

- Розмір турніру $k = 2$-7

- Більший $k \rightarrow$ сильніший тиск, швидша збіжність, але ризик передчасної збіжності

Оптимальні параметри залежать від конкретної задачі і часто визначаються експериментально.

Приклади успішного застосування генетичних алгоритмів

Щоб повністю оцінити практичну цінність генетичних алгоритмів, розглянемо детально декілька реальних кейсів, де вони продемонстрували впевнену перевагу над альтернативними методами.

1. Проектування антен для космічних місій NASA

У 2006 році NASA використала генетичні алгоритми для проектування антен для космічного апарату Space Technology 5. Це була місія з трьох

невеликих супутників для вивчення магнітосфери Землі. Антени повинні були задовольняти надзвичайно жорстким вимогам: працювати ефективно на трьох різних частотах (S-band для телеметрії, UHF для зв'язку), мати компактні розміри та малу вагу (через обмеження корисного навантаження), бути механічно стійкими до запуску та умов космосу (вібрації, вакуум, температурні перепади), забезпечувати достатнє підсилення та правильну поляризацію сигналу.

Простір можливих форм антен практично нескінченний. Традиційне проектування антен базується на фізичних принципах та інженерній інтуїції - інженер створює прототип, моделює його характеристики електромагнітним моделюванням, аналізує результати, модифікує дизайн і повторює цикл. Для складних багаточастотних антен цей процес може тривати місяці, і немає гарантії знаходження оптимального дизайну.

Дослідники з NASA Ames Research Center вирішили застосувати генетичний алгоритм. Антена представлялася як дерево гілок, де кожна гілка характеризується параметрами: довжина, кут відгалуження від батьківської гілки, товщина провідника, матеріал. Генотип був змішаним - частково дерево (структура гілок), частково дійсні числа (параметри кожної гілки), частково дискретні змінні (вибір матеріалу).

Функція пристосованості була багатокритеріальною і обчислювалася через повне електромагнітне моделювання методом моментів (Method of Moments). Кожне обчислення займало від двох до п'яти хвилин на робочій станції. Критерії включали: коефіцієнт підсилення на цільових частотах (максимізувати), коефіцієнт стоячої хвилі VSWR (мінімізувати, ідеально близько до 1), ширина смуги пропускання (достатня для передачі даних), ефективність випромінювання (максимізувати), поляризація (кругова, ліва або права за специфікацією), фізичні обмеження (вага менше двохсот грамів, розміри в межах доступного простору на супутнику).

Популяція складалася з ста антен.

Селекція - турнірна з розміром турніру п'ять.

Кросовер - спеціалізований для деревоподібних структур, обмінював піддерева між батьками.

Мутація включала: додавання або видалення гілки, зміну параметрів існуючої гілки (довжина, кут), заміну матеріалу.

Після п’ятдесяти поколінь (приблизно два тижні обчислень на кластері з десяти машин) генетичний алгоритм знайшов незвичайну “біоморфну” форму антени, яка нагадувала згорнуту руку з кількома пальцями різної довжини, спрямованими в різні сторони. Жоден інженер ніколи не запропонував би такої форми свідомо - вона виглядала хаотично і не відповідала стандартним принципам дизайну антен. Проте електромагнітне моделювання показало, що ця форма забезпечує чудові характеристики на всіх трьох цільових частотах завдяки резонансним ефектам різних гілок різної довжини.

Порівняння з традиційним дизайном показало: антена ГА була на двадцять відсотків ефективніша за найкращий ручний дизайн, час розробки скоротився з шести місяців до двох тижнів, дизайн успішно пройшов всі тести і був запущений у космос в 2006 році, працював бездоганно протягом всієї місії. Цей успіх продемонстрував, що еволюційні алгоритми можуть знаходити принципово нові, неочевидні розв’язок інженерних задач, які людина не розглянула б через їхню нестандартність.

2. Маршрутизація транспортних засобів для логістичної компанії

Велика логістична компанія в Європі, що спеціалізується на доставці товарів роздрібним магазинам, стикнулася з проблемою ефективного планування маршрутів своїх двохсот вантажівок для обслуговування п’яти тисяч клієнтів щоденно. Задача є узагальненням класичної задачі комівояжера і називається Vehicle Routing Problem with Time Windows - задача маршрутизації транспортних засобів з часовими вікнами.

Кожен клієнт має замовлення певного обсягу, який займає місце у вантажівці. Кожна вантажівка має обмежену місткість (зазвичай двадцять палет) і може виїхати лише з одного розподільчого центру, обслужити множину клієнтів і повернутися в центр до кінця робочого дня. Кожен клієнт має часове вікно - інтервал часу, коли він може прийняти доставку. Наприклад, супермаркет може приймати доставки лише з 6:00 до 9:00 ранку, до відкриття магазину. Якщо вантажівка приїжджає раніше, вона повинна чекати. Якщо пізніше - це порушення контракту зі штрафом або навіть відмовою від прийняття товару.

Цільові функції багатокритеріальні: мінімізувати загальну відстань (економія палива, зниження зносу транспорту), мінімізувати кількість використовуваних вантажівок (зменшення витрат на персонал), максимізувати дотримання часових вікон (задоволеність клієнтів,

уникнення штрафів), балансувати навантаження між водіями (справедливість, уникнення перевтоми).

Обмеження складні: місткість кожної вантажівки не може бути перевищена, загальний час маршруту обмежений (робочий день водія), часові вікна клієнтів повинні бути дотримані, деякі клієнти можуть вимагати спеціалізованих вантажівок (рефрижератори), дорожні обмеження (деякі вулиці закриті для великих вантажівок).

Простір розв'язків величезний. Якщо ми маємо двісті вантажівок і п'ять тисяч клієнтів, кількість способів розподілити клієнтів по вантажівках і визначити порядок їх відвідування астрономічна. Навіть якщо ми зафіксуємо розподіл клієнтів по вантажівках (кожна вантажівка отримує в середньому двадцять п'ять клієнтів), кількість можливих порядків для однієї вантажівки - факторіал двадцяти п'яти, а загальна кількість комбінацій - добуток двохсот факторіалів. Повний перебір абсолютно неможливий.

Дослідники розробили спеціалізований генетичний алгоритм з гібридними операторами. Кодування використовувало двокомпонентне представлення: гігантський тур - перестановка всіх п'яти тисяч клієнтів, розділювачі або декодер, що розбиває цей тур на двісті окремих маршрутів на основі обмежень місткості та часу.

Декодування працювало так: починаємо з порожньої першої вантажівки, йдемо по гігантському туру послідовно, додаємо клієнтів до поточної вантажівки. Якщо додавання наступного клієнта порушить обмеження місткості або часу, закриваємо поточний маршрут, починаємо наступну вантажівку. Це перетворює перестановку в набір маршрутів.

Функція пристосованості враховувала всі критерії через зважену суму:

$f = w_1 \cdot TotalDistance + w_2 \cdot NumberOfTrucks \cdot FixedCost + w_3 \cdot TimeWindowViolations + w_4 \cdot CapacityViolations$.

Ваги підбиралися так, щоб відображати реальні економічні пріоритети компанії.

Спеціалізовані оператори включали кросовер з частковим відображенням для перестановок, локальну оптимізацію 2-opt всередині кожного маршруту після декодування, реаллокацію клієнтів між сусідніми маршрутами (якщо два маршрути географічно близькі, спробувати перемістити клієнтів між ними для покращення), мутацію через інверсію сегментів та обмін клієнтів.

Алгоритм запускався з популяцією сто особин протягом тисячі поколінь, що займало приблизно тридцять хвилин на сервері. Це прийнятно для щоденного планування - оптимізація виконувалася вночі для наступного робочого дня.

Результати порівняння з попереднім рішенням (комбінація жадібних евристик та ручного коригування) показали: базовий метод давав загальну відстань сто двадцять п’ять тисяч кілометрів на день, використовував сто дев’яносто п’ять вантажівок, мав вісім відсотків порушень часових вікон. Генетичний алгоритм дав загальну відстань сто вісім тисяч кілометрів (економія тринадцять з половиною відсотків), використовував сто вісімдесят вантажівок (економія п’ятнадцять вантажівок), три відсотки порушень часових вікон (значне покращення задоволеності клієнтів).

Економічний ефект був значним: економія палива приблизно дві тисячі доларів на день, зменшення кількості вантажівок і водіїв заощадило близько п’яти тисяч доларів на день на зарплатах та обслуговуванні, зменшення порушень часових вікон знизило штрафи та покращило репутацію компанії. Сумарна річна економія склала понад два мільйони доларів.

Ключові фактори успіху генетичного алгоритму в цій задачі: здатність обробляти багатокритеріальні цілі природним чином, ефективне дослідження величезного простору комбінаторних рішень, легка інтеграція з локальними евристиками через гібридизацію, гнучкість до змін обмежень та додавання нових вимог клієнтів, розумний час обчислень, що дозволяє щоденне використання.

3. Планування навчального розкладу в університеті

Складання розкладу занять для великого університету є класичною задачею комбінаторної оптимізації з величезною кількістю обмежень. Розглянемо конкретний випадок університету в Західній Європі, який має п’ятдесят освітніх програм, триста викладачів, вісім тисяч студентів та триста аудиторій різних типів. Потрібно скласти розклад на семестр тривалістю п’ятнадцять тижнів, де щотижня проводиться близько трьох тисяч занять.

Задача має два типи обмежень: жорсткі та м’які. Жорсткі обмеження не можуть бути порушені ні за яких обставин, це фізичні або організаційні неможливості. До них належать: викладач не може вести дві пари одночасно в різних місцях, група студентів не може відвідувати дві пари одночасно, аудиторія не може бути зайнята двома заняттями одночасно,

спеціалізовані заняття (лабораторні роботи з фізики, хімічні практикуми) вимагають спеціалізованих аудиторій з необхідним обладнанням, місткість аудиторії повинна бути достатньою для кількості студентів на занятті.

М’які обмеження є побажаннями, їх порушення небажано, але допустимо. Ці обмеження оцінюються штрафами різної ваги залежно від їхньої важливості: мінімізація “вікон” у розкладі студентів (студенти не повинні мати великих перерв між заняттями в один день), рівномірний розподіл навантаження викладачів по днях тижня (викладач не повинен мати п’ять пар у понеділок і жодної в середу), уникнення занять у пізній час (після вісімнадцятої години), для деяких викладачів вікових або з особливими потребами, враховування побажань викладачів щодо днів та часу проведення занять, мінімізація переміщень між далекими корпусами (якщо у студента дві пари підряд, бажано, щоб вони були в одному або сусідніх будинках), рівномірний розподіл важких предметів по тижню (не ставити всі складні іспити математики та фізики в один день).

Простір розв’язків для такої задачі має розмірність, що перевищує усяку уяву. Кожне з трьох тисяч занять може бути призначене на один з приблизно двохсот можливих часових слотів (п’ять днів по вісім слотів на день) та одну з трьохсот аудиторій. Навіть якщо враховувати лише основні обмеження, кількість допустимих комбінацій настільки велика, що вичерпний пошук неможливий навіть за час, що перевищує вік Всесвіту.

Дослідники з факультету інформатики університету розробили генетичний алгоритм зі складною структурою кодування та спеціалізованими операторами.

Розв’язок представлється як матриця призначень розміром три тисячі на три, де кожен рядок відповідає одному заняттю, а три колонки містять: день тижня (понеділок=1, … п’ятниця=5), часовий слот у цей день (1-8), номер аудиторії (1-300).

Наприклад, рядок матриці для заняття номер двадцять п’ять може бути: (3,4,157), що означає середа, четвертий слот (приблизно 12:00-13:30), аудиторія номер сто п’ятдесят сім.

Функція пристосованості була складною багатокомпонентною конструкцією. Спочатку перевірялися всі жорсткі обмеження і підраховувалася кількість їх порушень. Для кожного типу порушення застосовувався дуже великий штраф:

$$f_{hard} = -10000 \cdot N_{teacher\_conflict} - 10000 \cdot N_{student\_conflict} - 10000 \cdot N_{room\_conflict} - 5000 \cdot N_{wrong\_room\_type} - 3000 \cdot N_{capacity\_exceeded}$$

Потім, якщо розв'язок було допустимим (всі жорсткі обмеження виконані), обчислювалися штрафи за порушення м'яких обмежень:

$$f_{soft} = -100 \cdot N_{student\_gaps} - 50 \cdot N_{teacher\_imbalance} - 30 \cdot N_{late\_classes} - 20 \cdot N_{teacher\_preferences} - 40 \cdot N_{long\_distances}$$

Загальна пристосованість: $f_{total} = f_{hard} + f_{soft}$. Мета - максимізувати цю функцію, тобто мінімізувати порушення.

Ініціалізація популяції не була повністю випадковою, оскільки чисто випадкові призначення майже завжди порушують жорсткі обмеження. Замість цього використовувалася евристична ініціалізація: тридцять відсотків популяції генерувалися жадібним конструктивним алгоритмом, який послідовно призначав кожному заняттю найкращий доступний слот з урахуванням вже призначених занять. Це давало допустимі або майже допустимі початкові рішення. Сімдесят відсотків популяції генерувалися випадково, але з обмеженнями - для кожного заняття вибирався випадковий слот серед тих, що не створюють очевидних конфліктів.

Спеціалізовані генетичні оператори були критично важливими для успіху. Стандартні оператори для такої задачі працювали б погано, оскільки вони руйнували б допустимість розв'язків. Кросовер використовував ідею repair-based crossover: два батьки схрещувалися стандартним одноточковим або двоточковим кросовером, що давало нащадка з можливими конфліктами. Потім застосовувалася процедура відновлення допустимості.

Процедура відновлення працювала таким чином: ідентифікувати всі заняття, що беруть участь у конфліктах (наприклад, два заняття в одній аудиторії одночасно), для кожного конфліктного заняття спробувати знайти вільний слот поблизу поточного часу, якщо вільний слот знайдено, переназначити заняття туди, якщо вільних слотів немає, спробувати обмін з іншим заняттям (swap), який би розв'язав конфлікт. Якщо після всіх спроб конфлікти залишаються, залишити розв'язок з конфліктами, але з великим штрафом - еволюція поступово їх усуне.

Мутація також була спеціалізованою і мала кілька типів. Випадкове переприсвоєння: з ймовірністю п'ять відсотків випадково обране заняття отримує новий випадковий слот. Спрямована мутація: з ймовірністю десять відсотків обирається заняття, що найбільше порушує м'які обмеження (наприклад, створює велике вікно в розкладі студентів), і для

нього шукається кращий слот. Обмін слотів: два випадкові заняття обмінюються своїми призначеннями, якщо це не створює нових конфліктів. Часова мутація: для випадкового заняття змінюється лише часовий слот, але не аудиторія, або навпаки.

Локальний пошук застосовувався після генетичних операцій до найкращих десяти відсотків популяції. Використовувався жадібний локальний пошук, що послідовно намагався покращити призначення кожного заняття: для кожного заняття в поточному рішенні, розглянути всі можливі альтернативні слоти, обчислити зміну в функції пристосованості для кожної альтернативи, якщо знайдено покращення, виконати перепризначення, повторювати до збіжності (коли жодне перепризначення не покращує рішення).

Популяція складалася зі ста особин, які еволюціонували протягом п’ятисот поколінь. Селекція - турнірна з розміром турніру сім.

Елітизм - п’ять найкращих особин автоматично переходили в наступне покоління. Алгоритм виконувався на сервері з шістнадцятьма ядрами, обчислення займали приблизно двадцять п’ять хвилин.

Результати порівняння з альтернативними методами були вражаючими. Ручне складання розкладу командою з трьох досвідчених адміністраторів займало близько сорока годин і все одно містило помилки та субоптимальності. Фінальний ручний розклад мав вісім конфліктів жорстких обмежень (які потім вимагали спеціальних домовленостей з викладачами для розв’язання) та двісті сорок сім порушень м’яких обмежень.

Жадібний конструктивний алгоритм, що послідовно призначав кожне заняття найкращому доступному слоту, працював п’ять хвилин і давав розклад без жорстких конфліктів, але зі сто вісімдесятьма дев’ятьма порушеннями м’яких обмежень.

Метод цілочисельного програмування, що моделював задачу як величезну задачу лінійної оптимізації з бінарними змінними, теоретично міг знайти оптимальний розклад, але потребував більше шести годин обчислень і все одно не збігся до оптимуму через величезну розмірність задачі. Результат після шести годин: нуль жорстких конфліктів, сто сорок п’ять порушень м’яких обмежень.

Генетичний алгоритм за двадцять п’ять хвилин дав нуль жорстких конфліктів (гарантована допустимість) та сто двадцять вісім порушень

м’яких обмежень (найкраще розв’язок серед усіх методів). Ці сто двадцять вісім порушень включали: сорок вікон у розкладах студентів (неможливо уникнути повністю через складність комбінацій), тридцять випадків дещо нерівномірного навантаження викладачів, двадцять три заняття у не найбільш бажаний час, тридцять п’ять випадків невиконання індивідуальних побажань викладачів.

Додаткові переваги генетичного алгоритму включали можливість інтерактивної роботи: алгоритм можна зупинити в будь-який момент і отримати найкраще поточне рішення, яке майже напевно є допустимим. Легку адаптацію до змін: якщо в останній момент викладач повідомляє, що не може вести пари в середу, це просто додається як нове жорстке обмеження, і алгоритм перезапускається. Гнучкість до нових типів обмежень: якщо університет вводить нову політику, наприклад, “лекції з однієї дисципліни не повинні йти два дні підряд”, це легко додати як нове м’яке обмеження зі своєю вагою.

Університет впровадив цю систему і використовує її вже більше десяти років для автоматизованого складання розкладу кожного семестру. Адміністратори лише задають початкові дані та коригують ваги м’яких обмежень відповідно до поточних пріоритетів, а алгоритм генерує розклад. Це заощадило тисячі годин ручної праці та значно покращило якість розкладів з точки зору задоволеності як студентів, так і викладачів.

4. Оптимізація портфеля фінансових інвестицій з реальними обмеженнями

Задача формування інвестиційного портфеля є однією з класичних задач математичних фінансів, формалізованою Гаррі Марковіцем у 1952 році, за що він отримав Нобелівську премію з економіки. Проте класична модель Марковіца робить кілька спрощувальних припущень, які рідко виконуються на практиці. Розглянемо реальний випадок інвестиційної компанії, яка керує портфелем обсягом п’ятдесят мільйонів доларів і має доступ до п’ятисот різних активів (акції, облігації, ETF, товарні ф’ючерси, валюти).

Класична задача Марковіца формулюється як: знайти вектор ваг $\mathbf{w} = (w_1, \dots, w_n)$, де w_i - частка капіталу, інвестована в актив i, що максимізує очікувану прибутковість при заданому рівні ризику або мінімізує ризик при заданій прибутковості. Математично це записується як квадратична задача оптимізації:

$$\min_{\mathbf{w}} \quad \mathbf{w}^T \boldsymbol{\Sigma} \mathbf{w} - \lambda \mathbf{w}^T \boldsymbol{\mu}$$

при обмеженнях:

$$\sum_{i=1}^{n} w_i = 1, \quad w_i \geq 0$$

де **μ** - вектор очікуваних прибутковостей активів, **Σ** - матриця коваріацій (міра спільної мінливості прибутковостей), що представляє ризик, λ - параметр, що контролює компроміс між прибутковістю та ризиком.

Проте реальна задача значно складніша через додаткові обмеження та цілі. Обмеження диверсифікації: не більше десяти відсотків капіталу в один актив (запобігання надмірній концентрації ризику), не більше тридцяти відсотків в один сектор економіки (технології, фінанси, охорона здоров'я тощо), мінімум п'ять різних секторів повинні бути представлені. Обмеження ліквідності: принаймні сорок відсотків портфеля повинно бути в високоліквідних активах, які можна швидко продати без значних втрат. Транзакційні витрати: при зміні структури портфеля сплачуються комісії брокеру та спред між ціною покупки та продажу, типово від нуля цілих одного десятого до нуля цілих п'яти десятих відсотка від суми угоди. Це створює нелінійність та робить часті зміни портфеля економічно невигідними.

Податкові міркування: у багатьох юрисдикціях прибуток від продажу активів оподатковується, причому ставка може залежати від терміну володіння (короткострокові або довгострокові інвестиції). Це створює стимул утримувати прибуткові позиції довше. ESG-критерії (Environmental, Social, Governance): сучасні інвестори все частіше вимагають врахування екологічних та соціальних факторів. Наприклад, виключення компаній з високими викидами CO2, компаній тютюнової або оборонної промисловості. Кардинальність: практичні міркування (складність моніторингу, адміністративні витрати) обмежують кількість різних активів у портфелі, наприклад, не більше п'ятдесяти позицій.

Багатокритеріальність: замість одного скалярного критерію якості інвестори хочуть одночасно максимізувати очікувану прибутковість, мінімізувати ризик (волатильність), мінімізувати максимальне просідання (drawdown - найбільше падіння від піку), максимізувати коефіцієнт Шарпа (відношення надлишкової прибутковості до волатильності), забезпечити певний рівень доходу (дивіденди, купони).

Ці складності роблять задачу змішаною цілочисельною нелінійною багатокритеріальною оптимізацією, для якої не існує ефективних точних

алгоритмів. Дослідники застосували багатокритеріальний генетичний алгоритм NSGA-II.

Кодування розв'язок було гібридним. Бінарна частина $\mathbf{b} = (b_1, \dots, b_n)$, де $b_i \in \{0,1\}$ вказує, чи включений актив i до портфеля. Неперервна частина $\mathbf{w} = (w_1, \dots, w_n)$, де $w_i \in [0,0.1]$ - вага активу в портфелі. Якщо $b_i = 0$, то w_i автоматично встановлюється в нуль незалежно від його значення. Після генетичних операцій ваги нормалізуються: $w_{i'} = w_i / \sum_{j:b_j=1} w_j$, щоб їхня сума дорівнювала одиниці.

Функції пристосованості (критерії для мінімізації) були:

$$f_1(\mathbf{w}) = -\mathbf{w}^T\boldsymbol{\mu} \quad \text{(мінімізувати для максимізації прибутку)}$$

$$f_2(\mathbf{w}) = \mathbf{w}^T\boldsymbol{\Sigma}\mathbf{w} \quad \text{(мінімізувати ризик)}$$

$$f_3(\mathbf{w}) = -\frac{\mathbf{w}^T\boldsymbol{\mu} - r_f}{\sqrt{\mathbf{w}^T\boldsymbol{\Sigma}\mathbf{w}}} \quad \text{(мінімізувати для максимізації коефіцієнта Шарпа)}$$

де r_f - безризикова ставка (дохідність державних облігацій).

Обмеження обробляються через штрафні функції, які додаються до кожної цільової функції: $f_{i'} = f_i + P \cdot V$, де V - сумарне порушення обмежень.

Генетичні оператори були адаптовані до змішаного кодування. Кросовер для бінарної частини - рівномірний, для неперервної частини - SBX. Спочатку застосовується кросовер окремо до бінарних та неперервних частин обох батьків, потім результати комбінуються. Мутація для бінарної частини - bit-flip з ймовірністю один через n, для неперервної частини - полінноміальна мутація. Після кожного оператора виконується нормалізація ваг та обрізання до допустимих меж.

Популяція складалася з двохсот особин, еволюція тривала п'ятдесят поколінь. Це займало приблизно дванадцять хвилин на робочій станції. NSGA-II використовував сортування за недомінованістю для селекції: популяція розбивається на фронти, де перший фронт містить Парето-оптимальні рішення, другий - рішення, оптимальні після видалення першого фронту, і так далі. Всередині кожного фронту підтримується різноманітність через crowding distance - метрику, що віддає перевагу рішенням у менш щільних регіонах простору критеріїв.

Експерименти проводилися на реальних історичних даних ринку за період 2015-2020 років. Використовувалися щоденні ціни закриття для п'ятисот активів з індексу S&P 500. Очікувані прибутковості оцінювалися як середні

історичні прибутковості за попередній рік з корекцією на тренди. Матриця коваріацій обчислювалася з історичних даних за той же період.

Результати показали, що алгоритм багатокритеріальної оптимізації знайшов фронт Парето з приблизно ста оптимальних портфелів, що представляють різні компроміси між критеріями. Кілька характерних точок фронту:

Консервативний портфель: річна очікувана прибутковість вісім з половиною відсотків, волатильність дванадцять відсотків (низький ризик), коефіцієнт Шарпа нуль цілих сімдесят одна сота, сорок два активи в портфелі, шістдесят відсотків в облігаціях та стабільних акціях, максимальна вага одного активу вісім відсотків.

Помірний портфель: прибутковість дванадцять цілих три десятих відсотка, волатильність вісімнадцять відсотків, коефіцієнт Шарпа нуль цілих шістдесят вісім сотих, тридцять п’ять активів, збалансована суміш акцій зростання та вартості, максимальна вага десять відсотків.

Агресивний портфель: прибутковість шістнадцять цілих сім десятих відсотка, волатильність двадцять шість відсотків (високий ризик), коефіцієнт Шарпа нуль цілих шістдесят чотири сотих, двадцять вісім активів, домінують технологічні акції та акції зростання, більша концентрація - кілька позицій по десять відсотків.

Порівняння з класичними методами показало переваги генетичного підходу. Квадратичне програмування (стандартний метод для моделі Марковіца) знаходить один оптимум для фіксованого значення параметра компромісу λ за приблизно п’ять хвилин. Щоб побудувати фронт Парето, потрібно багато запусків з різними λ, загальний час близько двох годин для ста точок. Крім того, стандартне квадратичне програмування погано обробляє складні обмеження (кардинальність, секторні обмеження), які роблять задачу змішаною цілочисельною.

NSGA-II знаходить весь фронт Парето (сто точок) за один запуск протягом дванадцяти хвилин. Природно обробляє всі типи обмежень без необхідності складних трюків або релаксацій. Легко додати нові критерії або обмеження, просто модифікувавши функцію пристосованості.

Бектестинг (перевірка на історичних даних) показав практичну цінність. Портфелі з фронту Парето, сформовані на основі даних за 2015-2018 роки, тестувалися на даних за 2019-2020 роки.

Результати: консервативний портфель дав реальну прибутковість сім цілих вісім десятих відсотка з волатильністю тринадцять цілих дві десятих відсотка (близько до прогнозу). Помірний портфель: одинадцять цілих п’ять десятих відсотка прибутковості, дев’ятнадцять відсотків волатильності. Агресивний портфель: п’ятнадцять цілих дві десятих відсотка прибутковості, двадцять вісім відсотків волатильності.

5. Налаштування гіперпараметрів глибоких нейронних мереж

У сучасному машинному навчанні глибокі нейронні мережі досягли вражаючих результатів у розпізнаванні зображень, обробці природної мови, грі в складні ігри. Проте їхня ефективність критично залежить від правильного вибору архітектури та гіперпараметрів. Розглянемо конкретну задачу: навчання згорткової нейронної мережі для класифікації зображень на датасеті CIFAR-100, який містить шістдесят тисяч кольорових зображень розміром тридцять два на тридцять два пікселі, розподілених по ста класах (різні види тварин, транспорту, побутових предметів).

Простір гіперпараметрів, що потребує оптимізації, величезний та гетерогенний. Архітектурні параметри включають: кількість згорткових блоків (шарів згортки з послідуючим pooling) від трьох до десяти, кількість фільтрів у кожному блоці - вибір з набору тридцять два, шістдесят чотири, сто двадцять вісім, двісті п’ятдесят шість, п’ятсот дванадцять, тип фільтрів - розмір ядра три на три, п’ять на п’ять, або сім на сім, тип pooling після кожної згортки - MaxPooling, AveragePooling, або взагалі без pooling, кількість повнозв’язних (fully-connected) шарів після згорткових - від одного до трьох, кількість нейронів у кожному FC-шарі від ста двадцяти восьми до двох тисяч сорока вісьми.

Параметри регуляризації: dropout rate (ймовірність вимкнення нейронів під час навчання для запобігання перенавчанню) від нуля до нуля цілих сімдесяти сотих, L2-регуляризація (weight decay) від нуля до нуля цілих одної сотої, batch normalization після кожної згортки - так чи ні.

Параметри навчання: швидкість навчання (learning rate) від нуля цілих одної десятитисячної до нуля цілих однієї десятої (логарифмічна шкала - має сенс пробувати значення, що відрізняються на порядки), тип оптимізатора - SGD (стохастичний градієнтний спуск), Adam, RMSprop, AdaGrad, для SGD - momentum від нуля до нуля цілих дев’яноста дев’яти сотих, для Adam - параметри beta1 та beta2, розмір batch (кількість зразків,

що обробляються одночасно) тридцять два, шістдесят чотири, сто двадцять вісім, або двісті п’ятдесят шість.

Параметри аугментації даних (штучне збільшення тренувальної вибірки через трансформації зображень): горизонтальне відображення - так чи ні, випадкове обрізання - так чи ні, обертання на випадковий кут від мінус п’ятнадцяти до плюс п’ятнадцяти градусів, зміна яскравості та контрасту в певних межах.

Якщо врахувати всі комбінації, простір пошуку містить понад 10^{15} різних конфігурацій. Більше того, оцінка кожної конфігурації (навчання мережі до збіжності) займає від двох до шести годин на потужному GPU, залежно від складності архітектури.

Дослідники з університету застосували генетичний алгоритм з спеціальними стратегіями для зменшення обчислювальних витрат. Кодування хромосоми було змішаним - комбінація цілочисельних, дійсних та категоріальних змінних. Для зручності всі параметри були відображені на інтервал [0,1] через відповідні функції масштабування, а потім декодовані назад перед навчанням мережі.

Наприклад, кількість згорткових блоків (ціле число від трьох до десяти) кодувалася як дійсне число $x \in [0,1]$, потім декодувалася: $n_{blocks} = 3 + \lfloor 7 \cdot x \rfloor$. Швидкість навчання (логарифмічна шкала) кодувалася як $x \in [0,1]$, декодування: $\eta = 10^{-4+3x}$, що дає діапазон від 10^{-4} до 10^{-1}.

Приклад хромосоми (спрощено, показано лише частину параметрів):

```
[n_blocks=0.6 (→7 блоків),
 filters=[0.4,0.6,0.7,0.8,0.9,1.0,1.0] (→[64,128,128,256,256,512,512]
),
 kernel_sizes=[0.2,0.2,0.6,0.4,...] (→[3,3,5,3,...]),
 fc_layers=0.5 (→2 FC шари),
 fc_neurons=[0.7,0.4] (→[512,256]),
 dropout=0.35,
 learning_rate=0.45 (→0.001),
 optimizer=1 (→Adam),
 batch_size=0.5 (→128)]
```

Функція пристосованості була багатокритеріальною, оскільки потрібен компроміс між точністю, складністю моделі та часом навчання:

$$f = \alpha \cdot Accuracy_{val} - \beta \cdot \frac{N_{parameters}}{10^6} - \gamma \cdot \frac{T_{train}}{hour}$$

де $Accuracy_{val}$ - точність на валідаційній вибірці, $N_{parameters}$ - загальна кількість параметрів мережі (ваг), T_{train} - час навчання. Ваги обиралися як $\alpha = 100,\ \beta = 1,\ \gamma = 2$, віддаючи пріоритет точності, але штрафуючи за надмірну складність.

Критичною проблемою був час обчислень. Якщо популяція з п’ятдесяти особин еволюціонує протягом п’ятдесяти поколінь, це дві тисячі п’ятсот оцінок функції пристосованості. При середньому часі чотири години на оцінку це десять тисяч годин або понад рік безперервних обчислень на одному GPU. Навіть з п’ятдесятьма GPU паралельно це займе тиждень.

Для вирозв’язок цієї проблеми використовувалися кілька стратегій прискорення. Early stopping: якщо після десяти епох навчання точність на валідаційній вибірці менша за тридцять відсотків (що значно нижче за випадковий вгадування для ста класів), зупиняємо навчання і призначаємо низьку пристосованість. Це дозволяє швидко відкидати явно погані архітектури.

Прогресивне навчання: перші двадцять поколінь всі мережі навчаються лише двадцять епох замість повних ста. Це дає грубу оцінку якості за значно менший час. Покоління двадцять один - сорок: кращі п’ятдесят відсотків особин донавчаються до п’ятдесяти епох. Останні десять поколінь: топ двадцять відсотків донавчаються до повних ста епох для точної оцінки.

Трансферне навчання: якщо нащадок має архітектуру, подібну до одного з батьків (наприклад, відрізняється лише в останніх шарах), ініціалізуємо ваги нащадка вагами батька де можливо, замість випадкової ініціалізації. Це значно прискорює навчання нащадка, оскільки перші шари вже навчені розпізнавати базові ознаки.

Генетичні оператори були спроектовані з урахуванням структури задачі. Кросовер був блоковим: архітектура мережі розглядалася як послідовність блоків (згортковий блок, FC-блок, блок параметрів навчання). Кросовер обмінював цілі блоки між батьками. Наприклад, нащадок міг успадкувати згорткову частину від першого батька (яка добре витягує ознаки) і FC-частину від другого батька (яка ефективно класифікує).

Мутація мала кілька типів відповідно до типу параметрів. Архітектурна мутація: з ймовірністю п’ять відсотків додати або видалити згортковий блок, з ймовірністю десять відсотків змінити кількість фільтрів у випадковому блоці (подвоїти або вдвічі зменшити), з ймовірністю п’ять відсотків змінити розмір ядра в блоці. Гіперпараметрична мутація: для

неперервних параметрів (dropout, learning rate) застосовувалася гаусівська мутація з малим стандартним відхиленням, для дискретних (optimizer, batch size) - рівномірна заміна на інше допустиме значення.

Експерименти проводилися на кластері з двадцяти чотирьох потужних GPU (NVIDIA V100) протягом трьох днів. Загальний обчислювальний бюджет: дві тисячі GPU-годин. Популяція - сорок особин, п’ятдесят поколінь. Завдяки стратегіям прискорення вдалося вмістити всі обчислення в заданий бюджет.

Результати порівняння з базовими методами та альтернативними підходами показали значні переваги генетичного алгоритму.

Baseline - ResNet-50, стандартна архітектура, розроблена експертами: точність на валідаційній вибірці сімдесят шість цілих три десятих відсотка, двадцять п’ять мільйонів параметрів, час навчання чотири години.

Random Search: випадкова вибірка тисячі конфігурацій з простору гіперпараметрів, навчання кожної: найкраща конфігурація дала сімдесят вісім цілих одну десяту відсотка точності, вісімнадцять мільйонів параметрів, п’ять годин навчання. Проте Random Search вимагав оцінки тисячі конфігурацій, що при паралелізації на двадцять чотири GPU зайняло приблизно сто п’ятдесят годин.

Bayesian Optimization: використання гаусівських процесів для моделювання функції пристосованості та розумного вибору наступних точок для оцінки: після трьохсот оцінок найкраща конфігурація дала сімдесят дев’ять цілих чотири десятих відсотка точності, двадцять два мільйони параметрів, чотири з половиною години навчання. Час роботи - приблизно п’ятдесят годин.

Генетичний алгоритм: після п’ятдесяти поколінь найкраща знайдена архітектура дала вісімдесят одну цілих дві десятих відсотка точності (найкращий результат), шістнадцять мільйонів параметрів (найкомпактніша мережа), три цілих вісім десятих години навчання (найшвидше навчання). Загальний час роботи ГА - сімдесят дві години завдяки паралелізації та стратегіям прискорення.

Розглянуті приклади демонструють, що генетичні алгоритми є потужним та універсальним інструментом для розв’язання складних оптимізаційних задач у різноманітних областях - від аерокосмічної інженерії до фінансів,

від логістики до машинного навчання, від енергетики до освіти. Спільними факторами успіху в усіх цих застосуваннях є кілька ключових характеристик задач та властивостей генетичних алгоритмів.

По-перше, всі розглянуті задачі мають величезні простори пошуку з астрономічною кількістю можливих рішень, де повний перебір абсолютно непрактичний. Генетичні алгоритми ефективно орієнтуються в таких просторах, концентруючи пошук на перспективних регіонах без необхідності вичерпного дослідження.

По-друге, багато задач є багатокритеріальними з конфліктними цілями, де не існує єдиного оптимального рішення, а є множина Парето-оптимальних компромісів. Популяційна природа генетичних алгоритмів ідеально підходить для знаходження цієї множини за один прогон, на відміну від скалярних методів оптимізації.

По-третє, задачі часто мають складні обмеження різних типів, нелінійні залежності, змішані типи змінних (дискретні та неперервні). Генетичні алгоритми природно працюють з такою гетерогенністю через гнучкість у виборі кодування та операторів.

По-четверте, у багатьох випадках функція якості є дорогою в обчисленні (CFD, FEA моделювання), зашумленою або навіть недиференційовною. Генетичні алгоритми не потребують градієнтної інформації і стійкі до шуму через популяційний підхід та статистичну природу селекції.

По-п’яте, генетичні алгоритми демонструють здатність знаходити неочевидні, креативні рішення, які людина-експерт не розглянула б через їхню нестандартність. Це особливо цінно в інженерному проектуванні та архітектурній оптимізації, де простір можливостей величезний, а людська інтуїція обмежена звичними шаблонами.

Водночас, успішне застосування генетичних алгоритмів вимагає вдумливого підходу: правильний вибір кодування та представлення рішень, розробка ефективних, часто спеціалізованих генетичних операторів для конкретної задачі, налаштування балансу між дослідженням та експлуатацією через параметри алгоритму, використання гібридизації з локальним пошуком або іншими методами де доцільно, застосування стратегій прискорення (метамоделювання, паралелізація, early stopping) для дорогих функцій, грамотна обробка обмежень через штрафи, відновлення або спеціальні оператори.

Контрольні запитання

28. Хто вважається батьком генетичних алгоритмів і в якому році була опублікована його фундаментальна робота?

29. Що таке теорема схем Холланда і яке її значення для теорії генетичних алгоритмів?

30. Поясніть концепцію “неявного паралелізму” (implicit parallelism) у генетичних алгоритмах.

31. Яку роль відіграв Девід Голдберг у розвитку та популяризації генетичних алгоритмів?

32. Що таке генетичне програмування і хто його розробив?

33. Опишіть основні етапи розвитку генетичних алгоритмів від 1960-х років до сьогодення.

34. Для яких типів задач генетичні алгоритми є найбільш ефективними?

35. Чому для задачі комівояжера з 50 містами повний перебір практично неможливий?

36. Поясніть різницю між задачами комбінаторної та неперервної оптимізації. Наведіть приклади.

37. Що таке множина Парето і чому генетичні алгоритми добре підходять для багатокритеріальної оптимізації?

38. У яких ситуаціях генетичні алгоритми НЕ є оптимальним вибором?

39. Наведіть приклади застосування генетичних алгоритмів у машинному навчанні.

40. Опишіть три основні механізми природної еволюції, які моделюють генетичні алгоритми.

41. Встановіть відповідність між біологічними термінами та поняттями генетичних алгоритмів (організм, хромосома, ген, алель, генотип, фенотип, пристосованість).

42. Поясніть, чому природний відбір можна розглядати як процес оптимізації.

43. Що таке схема (schema) у теорії Холланда? Чому короткі схеми низького порядку мають переваги?

44. Поясніть концепцію “будівельних блоків” (building blocks) у генетичних алгоритмах.

45. Як взаємодіють процеси дослідження (exploration) та експлуатації (exploitation) у генетичному алгоритмі?

46. Які основні типи кодування використовуються в генетичних алгоритмах?

47. Що таке проблема Гемінгової скелі (Hamming cliff) у бінарному кодуванні?

48. Як працює код Грея і чому він кращий за стандартне бінарне кодування для функцій неперервної оптимізації?

49. Наведіть формулу декодування бінарного рядка довжини k бітів у дійсне число з діапазону [a, b].

50. Опишіть переваги та недоліки дійсного кодування порівняно з бінарним.

51. Чому для задачі комівояжера потрібне перестановочне кодування?

52. Сформулюйте чотири принципи хорошого кодування: повнота, обґрунтованість, локальність, лаконічність.

53. У чому полягає суть оператора селекції і яка його роль у генетичному алгоритмі?

54. Поясніть принцип роботи пропорційної (рулетної) селекції. Які її основні недоліки?

55. Як працює ранг-селекція і які її переваги перед пропорційною?

56. Опишіть механізм турнірної селекції. Як розмір турніру k впливає на селекційний тиск?

57. Що таке елітизм і чому він важливий для збіжності генетичного алгоритму?

58. Сформулюйте теорему про збіжність генетичного алгоритму з елітизмом.

59. Яка основна ідея оператора кросинговеру?

60. Опишіть одноточковий кросинговер. Яка ймовірність руйнування схеми H з довжиною визначення $\delta(H)$?

61. Чим відрізняється двоточковий кросовер від одноточкового?

62. Поясніть принцип рівномірного кросинговеру. Коли він найбільш ефективний?

63. Як працює арифметичний кросовер для дійсного кодування?

64. Що таке SBX (Simulated Binary Crossover) і як параметр ηc впливає на поведінку оператора?

65. Опишіть Order Crossover (OX) для перестановок. Чому звичайні оператори не підходять для задачі комівояжера?

66. Як працює PMX (Partially Mapped Crossover)?

67. Що таке Edge Recombination Crossover і чому він ефективний для TSP?

68. Яка роль мутації в генетичному алгоритмі?

69. Опишіть бітову мутацію. Яка типова ймовірність мутації pm для бінарного кодування?

70. Як працює гаусівська мутація для дійсного кодування?

71. Поясніть різницю між гаусівською та рівномірною мутацією.

72. Що таке полінноміальна мутація і як параметр ηm впливає на величину змін?

73. Опишіть чотири типи мутації для перестановок: swap, insert, inversion, scramble.

74. Поясніть концепцію адаптивної мутації. Чому параметри можуть змінюватися з часом?

75. Що таке самоадаптація параметрів у еволюційних стратегіях?

76. Що таке меметичні алгоритми (гібридизація з локальним пошуком)?

77. Опишіть механізм fitness sharing для підтримання різноманітності.

78. Як обробляються обмеження в генетичних алгоритмах (штрафні функції, repair operators)?

79. Що таке коеволюція і як вона застосовується?

80. Опишіть покроково базовий генетичний алгоритм.

81. Які критерії зупинки використовуються в генетичних алгоритмах?

82. Як розмір популяції N впливає на ефективність алгоритму?

83. Які типові значення ймовірності кросинговеру pc?

84. Як вибирається ймовірність мутації pm?

Методичні вказівки до лабораторної роботи

Лабораторна робота: Реалізація та дослідження генетичного алгоритму

Мета роботи

Набути практичних навичок реалізації генетичного алгоритму, дослідити вплив параметрів алгоритму на якість та швидкість оптимізації, порівняти різні генетичні оператори.

Теоретичні відомості

Перед виконанням роботи студент повинен вивчити: 1. Основні поняття: популяція, хромосома, пристосованість, покоління 2. Структуру базового генетичного алгоритму 3. Оператори селекції, кросинговеру та мутації 4. Критерії зупинки алгоритму 5. Методи кодування рішень

Завдання

Частина 1: Реалізація базового алгоритму

85. Вибір мови програмування: Python (рекомендовано), MATLAB, C++, Java або інша

86. Реалізувати основні компоненти:

- Структуру даних для представлення популяції
- Функцію ініціалізації популяції
- Функцію обчислення пристосованості
- Оператор селекції (турнірна, розмір турніру k=3)
- Оператор кросинговеру (одноточковий для бінарного або арифметичний для дійсного кодування)
- Оператор мутації (бітова для бінарного або гаусівська для дійсного)
- Основний цикл еволюції
- Механізм збереження статистики (найкраща пристосованість за поколінням)

87. Тестова функція: Реалізувати функцію Растрігіна:

$$f(\mathbf{x}) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$$

Параметри:

- Розмірність: n = 2 (для візуалізації), потім n = 5, 10
- Область пошуку: $x_i \in [-5.12, 5.12]$
- Глобальний мінімум: $\mathbf{x}^* = (0,0,\ldots,0)$, $f(\mathbf{x}^*) = 0$

Частина 2: Базові експерименти

Експеримент 1: Вплив розміру популяції

Фіксовані параметри: - Кількість поколінь: T = 100 - Ймовірність кросинговеру: pc = 0.8 - Ймовірність мутації: pm = 0.1 - Розмірність: n = 5

Змінний параметр - розмір популяції N: - N = 20, 50, 100, 200, 500

Для кожного значення N виконати 30 незалежних запусків.

Що вимірювати: 1. Середнє значення найкращої знайденої пристосованості 2. Стандартне відхилення результатів 3. Відсоток успішних запусків (f < 1.0) 4. Середня кількість обчислень функції до збіжності 5. Час виконання

Що побудувати: - Графік залежності якості рішення від розміру популяції - Графік збіжності (найкраща пристосованість vs покоління) для різних N - Box plot розподілу результатів 30 запусків

Експеримент 2: Вплив ймовірності мутації

Фіксовані параметри: - N = 100 - T = 100 - pc = 0.8 - n = 5

Змінний параметр - ймовірність мутації pm: - pm = 0.01, 0.05, 0.1, 0.2, 0.5

Виконати те ж саме, що в Експерименті 1.

Експеримент 3: Вплив ймовірності кросинговеру

Фіксовані параметри: - N = 100 - T = 100 - pm = 0.1 - n = 5

Змінний параметр - ймовірність кросинговеру pc: - pc = 0.0, 0.3, 0.6, 0.8, 1.0

Експеримент 4: Вплив методу селекції

Порівняти три методи селекції: 1. Турнірна (k = 2) 2. Турнірна (k = 5) 3. Рулетна (пропорційна) 4. Ранг-селекція

Фіксовані параметри: N = 100, T = 100, pc = 0.8, pm = 0.1, n = 5

Частина 4: Візуалізація (для n = 2)

88. Побудувати 3D-поверхню функції Растрігіна

89. Показати початкову популяцію на контурному графіку

90. Анімувати еволюцію популяції (кожні 5-10 поколінь)

91. Показати фінальну популяцію біля глобального мінімуму

Структура звіту

92. Титульна сторінка: назва роботи, ПІБ, група, дата

93. Мета роботи (1 абзац)

94. Теоретичні відомості (1-2 сторінки):

- Структура генетичного алгоритму
- Опис використаних операторів
- Функція Растрігіна та її властивості

95. Опис реалізації (2-3 сторінки):

- Мова програмування та бібліотеки
- Структури даних

- Псевдокод основних функцій
- Особливості реалізації

96. Результати експериментів (5-7 сторінок):

Для кожного експерименту:

- Таблиця з числовими результатами
- Графіки (збіжності, залежності від параметрів, box plots)
- Аналіз результатів (2-3 абзаци)

Приклад таблиці для Експерименту 1:

N	Середнє f	Std	Успішних (%)	Обчислень	Час (с)
20	15.3	8.2	13.3	2000	0.5
50	5.7	4.1	46.7	5000	1.2
100	1.2	1.8	73.3	10000	2.5
200	0.3	0.5	93.3	20000	5.1
500	0.1	0.2	96.7	50000	13.2

97. Візуалізація (1-2 сторінки):

- 3D-графік функції
- Еволюція популяції (кілька кадрів)
- Контурний графік з траєкторіями

98. Порівняльний аналіз (1-2 сторінки):

- Зведена таблиця порівняння всіх експериментів
- Висновки про найкращі налаштування
- Порівняння з іншими методами (якщо виконано)

99. Висновки (0.5-1 сторінка):

- Які параметри найбільше впливають на ефективність
- Оптимальні налаштування для даної задачі
- Переваги та недоліки генетичного алгоритму
- Можливі напрямки покращення

100. Додатки:

- Повний вихідний код програми

- Додаткові графіки

Критерії оцінювання

Максимальна оцінка: 100 балів

101. Реалізація алгоритму (30 балів):

- Коректність реалізації (15 балів)
- Якість коду (читабельність, коментарі, структура) (10 балів)
- Ефективність реалізації (5 балів)

102. Експерименти (40 балів):

- Експеримент 1: Вплив розміру популяції (10 балів)
- Експеримент 2: Вплив ймовірності мутації (10 балів)
- Експеримент 3: Вплив ймовірності кросинговеру (10 балів)
- Експеримент 4: Порівняння методів селекції (10 балів)

103. Аналіз результатів (15 балів):

- Якість графіків та таблиць (5 балів)
- Глибина аналізу (7 балів)
- Висновки та рекомендації (3 бали)

104. Візуалізація (10 балів):

- 3D-графік функції (3 бали)
- Еволюція популяції (4 бали)
- Інші візуалізації (3 бали)

105. Оформлення звіту (5 балів):

- Структура та повнота (3 бали)
- Грамотність (2 бали)

Бонусні бали (до +20): - Порівняння з іншими методами оптимізації (+10) - Реалізація додаткових операторів (+5) - Анімація еволюції (+5) - Дослідження функцій вищої розмірності (n > 10) (+5)

Контрольні питання до захисту

106. Поясніть, як ви обрали параметри алгоритму для базових експериментів

107. Чому в ваших експериментах певний параметр виявився найбільш впливовим?

108. Як би ви покращили алгоритм для даної задачі?

109. Порівняйте ефективність різних методів селекції на вашій функції

110. Що відбувається, якщо ймовірність мутації занадто висока або занадто низька?

111. Чому генетичний алгоритм іноді застряє в локальних мінімумах функції Растрігіна?

112. Як змінюється поведінка алгоритму зі збільшенням розмірності задачі?

113. Поясніть графік збіжності вашого алгоритму

114. Які стратегії можна використати для прискорення алгоритму?

115. Як би ви адаптували ваш код для розв'язання реальної задачі оптимізації?

Тести (базовий рівень)

116. Хто є автором фундаментальної теорії генетичних алгоритмів і в якому році була опублікована його основна робота?

A) Алан Тюрінг, 1950

B) Джон Холланд, 1975

C) Девід Голдберг, 1989

D) Джон Коза, 1992

Правильна відповідь: B

6. Яка формула використовується для функції Растрігіна для n змінних?

A) $f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$

B) $f(\mathbf{x}) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$

C) $f(\mathbf{x}) = \sum_{i=1}^{n} |x_i|$

D) $f(\mathbf{x}) = \sum_{i=1}^{n} x_i^4$

Правильна відповідь: B

5. Що таке схема (schema) в термінах теорії Холланда?

A) Алгоритм роботи генетичного оператора

B) Шаблон, що описує підмножину рішень з фіксованими значеннями в деяких позиціях

C) Структура популяції

D) Графік збіжності алгоритму

Правильна відповідь: B

6. Яка типова ймовірність мутації pm для бінарної хромосоми довжини l?

A) pm = 0.5

B) pm = 1/l

C) pm = 0.9

D) pm = 1/N (де N - розмір популяції)

Правильна відповідь: B

8. Що таке елітизм у генетичному алгоритмі?

A) Вибір найгірших особин для мутації

B) Гарантоване збереження кількох найкращих особин у наступному поколінні

C) Метод ініціалізації популяції

D) Тип оператора кросинговеру

Правильна відповідь: B

7. У турнірній селекції з розміром турніру k=2, скільки особин випадково вибираються для порівняння?

A) 1

B) 2

C) Половина популяції

D) Вся популяція

Правильна відповідь: B

10. Що є головною перевагою коду Грея над стандартним бінарним кодуванням?

A) Використовує менше біт

B) Швидше декодується

C) Послідовні числа відрізняються лише в одному біті

D) Дозволяє кодувати від’ємні числа

Правильна відповідь: C

10. Яка формула декодує k-бітне бінарне число в дійсне значення y з діапазону [a, b]?

A) $y = a + (b - a) \cdot x$

B) $y = a + \frac{b-a}{2^k-1} \cdot \sum_{i=1}^{k} x_i \cdot 2^{k-i}$

C) $y = (a + b)/2 + x$

D) $y = a \cdot b \cdot x$

Правильна відповідь: B

10. Скільки приблизно локальних мінімумів має функція Растрігіна розмірності n на стандартній області?

A) n

B) 2n

C) n^2

D) 10^n

Правильна відповідь: D

11. Що відбувається в операторі одноточкового кросинговеру?

A) Один біт випадково змінюється

B) Хромосома розрізається в одній точці і частини батьків обмінюються

C) Вибирається одна особина для копіювання

D) Видаляється один ген

Правильна відповідь: B

Тести підвищеної складності

117. Теорема схем Холланда стверджує, що схеми з низьким порядком, короткою довжиною визначення та пристосованістю вище середньої:

A) Залишаються незмінними в популяції

B) Експоненційно зростають у популяції в наступних поколіннях

C) Зменшуються в популяції

D) Змінюються лінійно

Правильна відповідь: B

Пояснення: Теорема схем Холланда математично формулюється як:

$$E[m(H,t+1)] \geq m(H,t) \cdot \frac{f(H)}{f_{avg}} \cdot \left[1 - p_c \cdot \frac{\delta(H)}{l-1} - o(H) \cdot p_m\right]$$

Перший множник $\frac{f(H)}{f_{avg}}$ показує, що якщо схема має пристосованість вище середньої ($f(H) > f_{avg}$), цей множник більший за 1, що призводить до експоненційного зростання кількості екземплярів схеми в популяції. Другий множник у квадратних дужках показує, що схеми з низьким порядком $o(H)$ та короткою довжиною визначення $\delta(H)$ мають менші шанси бути зруйнованими генетичними операторами. Таким чином, “хороші будівельні блоки” - короткі схеми низького порядку з високою пристосованістю - швидко поширюються в популяції, що є основою ефективності генетичних алгоритмів.

7. У задачі комівояжера з 20 містами, скільки існує різних можливих маршрутів (без урахування напрямку обходу)?

A) 20!

B) (20-1)!/2

C) 2^{20}

D) 20^2

Правильна відповідь: B

Пояснення: Для задачі комівояжера (TSP) з n містами загальна кількість можливих перестановок міст дорівнює n!. Однак, оскільки: 1) Початкове місто можна фіксувати (це зменшує кількість до (n-1)!) 2) Маршрут в прямому та зворотному напрямку має однакову довжину (це ділить на 2)

Отримуємо формулу: $(n-1)!/2$

Для n=20 це дає: $19!/2 \approx 6 \times 10^{16}$ різних маршрутів. Якщо комп'ютер може перевірити мільярд маршрутів за секунду, то для повного перебору знадобиться понад 1900 років безперервних обчислень. Це демонструє, чому задача комівояжера є NP-важкою і чому потрібні евристичні методи, такі як генетичні алгоритми, для знаходження хороших (але не обов'язково оптимальних) рішень за розумний час.

6. Популяція розміру N=100 з бінарних рядків довжини l=50 неявно обробляє приблизно скільки різних схем паралельно (згідно з концепцією неявного паралелізму Холланда)?

A) 100

B) 5,000

C) 100,000

D) 1,000,000

Правильна відповідь: D

Пояснення: Концепція неявного паралелізму - одна з найважливіших теоретичних основ генетичних алгоритмів. Холланд показав, що типова популяція розміру N неявно оцінює приблизно $O(N^3)$ різних схем паралельно, навіть працюючи лише з N індивідуальними рішеннями.

Для N=100: $N^3 = 100^3 = 1{,}000{,}000$

Це відбувається тому, що кожна особина в популяції є екземпляром не однієї, а багатьох різних схем одночасно. Наприклад, бінарний рядок "101..." одночасно є екземпляром схеми "1...", схеми "*0*...", схеми "10*..." тощо. Через селекцію, коли особина з хорошою пристосованістю розмножується, автоматично розмножуються ВСІ схеми, екземпляром яких вона є. Це надзвичайно потужна властивість - алгоритм досліджує

величезну кількість підобластей простору пошуку (представлених схемами) без явного обчислення функції для кожної схеми окремо. Саме тому генетичні алгоритми можуть ефективно працювати в просторах з астрономічною кількістю варіантів.

7. Яка основна причина, чому Order Crossover (OX) використовується для задачі комівояжера замість стандартного одноточкового кросинговеру?

A) OX швидше виконується

B) Стандартний кросовер може створити недопустимі перестановки з дублікатами або пропущеними містами

C) OX завжди покращує рішення

D) Стандартний кросовер вимагає більше пам'яті

Правильна відповідь: B

Пояснення: У задачі комівояжера рішення представлено як перестановку міст, де кожне місто має з'явитися рівно один раз. Це критична властивість допустимості. Розглянемо приклад чому стандартний кросовер не працює:

Батько 1: (1 2 3 4 5 6)

Батько 2: (4 5 6 1 2 3)

Одноточковий кросовер з точкою 3:

Нащадок: (1 2 3 | 1 2 3) – недопустимо. Міста 1,2,3 повторюються, міста 4,5,6 відсутні.

Кросовер порядку спеціально розроблений для збереження властивості перестановки:

1. Копіює підрядок від одного батька (гарантуючи, що ці міста будуть присутні)

2. Заповнює решту позицій містами з іншого батька в порядку їх появи, пропускаючи вже використані

3. Результат завжди є коректною перестановкою

Більше того, OX має семантичне значення для TSP: він намагається зберегти хороші "підмаршрути" (послідовності міст) від обох батьків, що має сенс, оскільки якщо певна послідовність міст ефективна в одному турі, вона може

бути ефективною і в іншому. Це приклад того, як генетичні оператори повинні бути адаптовані до специфіки задачі для ефективної роботи.

9. У функції SBX (Simulated Binary Crossover) параметр розподілу ηc = 20. Що це означає для поведінки оператора?

A) Нащадки будуть дуже далеко від батьків

B) Нащадки будуть близько до батьків з невеликою варіацією

C) Кросовер не відбудеться

D) Параметр не впливає на результат

Правильна відповідь: B

Пояснення: Параметр η_c (distribution index) у SBX контролює форму розподілу ймовірностей для генерації нащадків. Він має обернений вплив на дисперсію:

- Великий η_c (наприклад, 15-30): Нащадки генеруються близько до батьків, розподіл має високий пік навколо батьківських значень. Це сприяє експлуатації - тонкому налаштуванню вже знайдених хороших рішень.
- Малий η_c (наприклад, 2-5): Нащадки можуть бути далеко від батьків, розподіл більш рівномірний. Це сприяє дослідженню - пошуку нових областей простору.

Математично, у SBX:

$$\beta_i = \begin{cases} (2u_i)^{1/(\eta_c+1)}, & u_i \le 0.5 \\ (1/(2(1-u_i)))^{1/(\eta_c+1)}, & u_i > 0.5 \end{cases}$$

При $\eta_c = 20$, показник степеня $1/21 \approx 0.048$ - дуже малий, що означає, що навіть великі відхилення u від 0.5 дають β близький до 1, тобто нащадки близькі до середнього батьківських значень.

Це важливо для пізніх етапів еволюції: коли популяція вже знайшла хороші регіони простору пошуку, велика η_c дозволяє "відполірувати" рішення, роблячи малі покращення без ризику випадкового стрибка в погані області. У NSGA-II, де SBX є стандартним оператором, типово використовують $\eta_c = 15 - 20$ саме з цієї причини.

8. Чому в початкових поколіннях генетичного алгоритму бажано мати високу різноманітність популяції, а в пізніх - низьку?

A) Це не має значення

B) На початку потрібно дослідження різних регіонів простору, пізніше - локальне покращення знайдених рішень

C) Висока різноманітність завжди краща

D) Низька різноманітність на початку прискорює збіжність

Правильна відповідь: B

Пояснення: Це фундаментальний принцип балансу між дослідженням та експлуатацією в еволюційних алгоритмах:

Рання фаза (дослідження):

- Популяція розсіяна по великій області простору пошуку

- Мета: знайти перспективні регіони, де можуть бути хороші рішення

- Висока різноманітність дозволяє уникнути передчасної збіжності до субоптимальних областей

- Аналогія: група дослідників розпорошується по великій території, щоб знайти найбільш обіцяючі місця

Пізня фаза (експлуатація):

- Популяція концентрується навколо кількох найкращих знайдених рішень

- Мета: покращити ці рішення, знайти локальні оптимуми в перспективних регіонах

- Низька різноманітність означає, що всі зусилля спрямовані на “відполірування” кращих варіантів

- Аналогія: дослідники збираються в найперспективніших місцях і ретельно їх вивчають

Механізми контролю:

1. Адаптивна мутація: $\sigma(t) = \sigma_0 \cdot e^{-\lambda t}$ - зменшення величини мутацій з часом

2. Зміна селекційного тиску: збільшення розміру турніру в пізніх поколіннях

3. Моніторинг різноманітності: $D(t) = \frac{1}{N(N-1)} \sum_{i,j} d\,(x_i, x_j)$

Якщо різноманітність занадто швидко падає (передчасна збіжність), алгоритм застряє в локальному оптимумі. Якщо вона залишається високою

надто довго, алгоритм повільно збігається і не використовує знайдену інформацію. Оптимальна траєкторія - поступовий перехід від дослідження до експлуатації.

11. У генетичному алгоритмі з популяцією N=100, яка частина популяції (в середньому) буде складатися з копій найкращої особини після одного покоління турнірної селекції з k=3, якщо ця особина значно краща за всіх інших?

A) 1% (1 особина)

B) ~3% (3 особини)

C) ~30% (30 особин)

D) 100% (вся популяція)

Правильна відповідь: C

Пояснення: Це питання перевіряє розуміння селекційного тиску в турнірній селекції. Проаналізуємо математично:

Ймовірність вибору найкращої особини в турнірі розміру k=3:

Якщо найкраща особина значно краща за всіх інших, вона виграє турнір, якщо потрапить до нього. Ймовірність НЕ потрапити в турнір з 3 учасників:

$$P(\text{не вибрана}) = \frac{99}{100} \cdot \frac{98}{99} \cdot \frac{97}{98} = \frac{97}{100} = 0.97$$

Ймовірність бути вибраною:

$$P(\text{вибрана}) = 1 - 0.97 = 0.03 \approx 3\%$$

Але це ймовірність для ОДНОГО турніру. Ми проводимо N=100 турнірів для формування батьківської популяції. Очікувана кількість разів, коли найкраща особина буде вибрана як батько:

$$E[\text{вибрана}] = 100 \cdot 0.03 = 3$$

Однак, якщо вона вибрана як батько 3 рази і кожного разу бере участь у кросинговері (можливо з іншими особинами), і якщо нащадки успадковують хороші властивості, в новому поколінні може бути більше схожих особин. Крім того, з елітизмом вона гарантовано переходить.

На практиці, через комбінацію:

- Прямих копій від елітизму (1-2 особини)

- Випадків, коли кросовер не відбувся і створилася копія (~10-15 особин)

- Нащадків, дуже схожих після кросинговеру (~10-15 особин)

Реально в наступному поколінні буде близько 25-35% особин, що є копіями або дуже близькими до найкращої. Це демонструє потужний селекційний тиск турнірної селекції з k=3. Для порівняння, з k=2 це було б ~15-20%, з k=5 - ~40-50%. Вибір k - це баланс між швидкою збіжністю (великий k) та підтримкою різноманітності (малий k).

8. Генетичний алгоритм з елітизмом ne=1 та достатньо високою ймовірністю мутації pm збігається до глобального оптимуму з ймовірністю 1 при t→∞. Яка основна математична причина цього?

A) Елітизм гарантує монотонне покращення

B) Мутація забезпечує ненульову ймовірність досягнення будь-якої точки простору з будь-якої іншої за скінченний час

C) Обидві умови A та B разом створюють ергодичний марковський ланцюг

D) Популяція збільшується з часом

Правильна відповідь: C

Пояснення: Це глибоке теоретичне питання про умови збіжності еволюційних алгоритмів. Розберемо детально:

Умова A (Елітизм): Елітизм забезпечує, що найкраще знайдене рішення ніколи не втрачається:

$$\max_i f_i(t+1) \geq \max_i f_i(t)$$

Це означає, що послідовність найкращих пристосованостей монотонно зростає і обмежена зверху (глобальним оптимумом), отже вона збігається. Але збіжність не обов'язково до глобального оптимуму - може бути до локального!

Умова B (Ергодичність через мутацію): Мутація з ймовірністю pm > 0 на кожному біті означає, що існує ненульова ймовірність перетворити будь-яке рішення в будь-яке інше за скінченну кількість кроків. Формально:

$$P(\text{досягти } x^* \text{ з } x \text{ за } k \text{ кроків}) > 0$$

для деякого скінченного k.

Це робить простір станів (популяцій) ергодичним - будь-який стан досяжний з будь-якого іншого.

Комбінація A+B: Разом ці дві властивості створюють ергодичний марковський ланцюг з абсорбуючим станом:

- Генетичний алгоритм можна моделювати як марковський процес, де стан - це поточна популяція

- Ергодичність означає, що ланцюг може досягнути будь-якого стану

- Елітизм означає, що якщо популяція досягла стану, що містить глобальний оптимум, вона ніколи не залишить цей "хороший" стан

- Оскільки є ненульова ймовірність мутувати в глобальний оптимум, і після цього він ніколи не втрачається, за нескінченний час ймовірність досягнення глобального оптимуму прямує до 1

Математично:

$$\lim_{t\to\infty} P(\exists i: x_i(t) = x^*) = 1$$

Практичне значення: Хоча це теоретично гарантує збіжність, швидкість може бути надзвичайно повільною. Якщо ймовірність мутувати в оптимум 10^{-50}, "нескінченний час" на практиці означає більше віку Всесвіту. Тому на практиці важливіші емпіричні результати швидкості збіжності, ніж асимптотичні гарантії. Це також пояснює, чому надто висока pm може погіршити практичну збіжність - вона робить процес більш схожим на випадковий пошук.

9. У задачі оптимізації портфеля з 500 активів використовується генетичний алгоритм. Чому змішане (бінарне + дійсне) кодування краще за чисто дійсне?

A) Змішане кодування завжди краще

B) Бінарна частина вказує, які активи включені (відбір активів), дійсна частина - їхні ваги, що природно моделює структуру задачі

C) Змішане кодування швидше обчислюється

D) Різниці немає

Правильна відповідь: B

Пояснення: Це приклад важливого принципу: кодування повинно відображати структуру задачі. Розглянемо детально:

Структура задачі оптимізації портфеля: Формально потрібно знайти вектор ваг $\mathbf{w} = (w_1, \ldots, w_{500})$, де: - $w_i \geq 0$ - вага активу і в портфелі - $\sum_{i=1}^{500} w_i = 1$ -

загальна сума ваг = 100% - Додаткові обмеження: наприклад, не більше 50 активів в портфелі (кардинальність)

Проблема чисто дійсного кодування:

$$\mathbf{x} = (w_1, w_2, \ldots, w_{500}) \in [0,1]^{500}$$

Проблеми:

1. Багато активів будуть мати дуже малі ваги (наприклад, 0.001%), що практично безглуздо

2. Обмеження кардинальності (максимум 50 активів) важко забезпечити

3. Після кросинговеру та мутації потрібна складна нормалізація для забезпечення $\sum w_i = 1$

4. Простір пошуку містить багато "майже однакових" рішень (портфель з активами {0.01, 0.02, 0.97} vs {0, 0.03, 0.97})

Переваги змішаного кодування:

$$\mathbf{x} = (\mathbf{b}, \mathbf{w}) = ((b_1, \ldots, b_{500}), (w_1, \ldots, w_{500}))$$

де: - $b_i \in \{0,1\}$ - бінарний індикатор: чи включений актив і - $w_i \in [0,1]$ - вага активу (якщо включений)

Декодування:

$$w'_i = \begin{cases} \dfrac{w_i}{\sum_{j:b_j=1} w_j}, & \text{якщо } b_i = 1 \\ 0, & \text{якщо } b_i = 0 \end{cases}$$

Переваги:

1. Природне відображення: "Які активи вибрати?" (дискретне рішення) + "Як розподілити капітал?" (неперервне рішення)

2. Легке забезпечення кардинальності: $\sum_{i=1}^{500} b_i \leq 50$ - просто підрахувати одиниці

3. Ефективніші генетичні оператори:

- Бітова мутація в **b** додає/видаляє активи - Гаусівська мутація в **w** змінює розподіл капіталу

- Можна використовувати різні оператори для різних частин хромосоми

4. Менший ефективний простір пошуку: замість розгляду 500-вимірного неперервного простору, розглядаємо ~50 активних змінних

5. Інтерпретованість: легко бачити, які активи вибрані

Генетичні оператори для змішаного кодування:

```
Кросинговер:
  Бінарна частина: Uniform crossover
  Дійсна частина: SBX

Мутація:
  Бінарна частина: Bit-flip (pm = 1/500)
  Дійсна частина: Polynomial (pm = 1/50 для активних)
```

Це демонструє загальний принцип: хороше кодування використовує знання про структуру задачі і робить генетичні оператори більш “осмисленими” з точки зору семантики задачі.

10. Чому для задачі комівояжера локальний пошук 2-opt часто комбінується з генетичним алгоритмом (меметичний алгоритм), хоча ГА сам є методом оптимізації?

A) Для прискорення обчислень

B) ГА добре знаходить перспективні регіони простору (хороші підмаршрути), але погано “відполіровує” рішення; 2-opt швидко покращує локально

C) 2-opt замінює мутацію

D) Це не має переваг

Правильна відповідь: B

Пояснення: Це фундаментальне питання про гібридні алгоритми та компліментарність різних методів оптимізації. Детальний аналіз:

Сильні сторони генетичного алгоритму:

1. Глобальний пошук: Популяція одночасно досліджує різні регіони простору

2. Комбінування хороших структур: Кросовер (особливо Order Crossover або Edge Recombination) може комбінувати хороші підмаршрути з різних турів

- Приклад: якщо батько 1 має ефективний підмаршрут A→B→C→D на півночі карти, а батько 2 має E→F→G→H на півдні, нащадок може успадкувати обидва 3. Стійкість до локальних оптимумів: Різноманітність популяції допомагає уникати застрягання

Слабкі сторони ГА для TSP:

1. Повільна локальна збіжність: Щоб перейти від туру довжини 1005 до туру довжини 1000 може знадобитися багато поколінь

2. Випадковість операторів: Мутація та кросовер не гарантують покращення - можуть погіршити тур

3. Великий простір пошуку: Для n=100 міст простір містить 10^{157} турів - складно знайти точний оптимум

Сильні сторони 2-opt локального пошуку: 2-opt систематично перевіряє можливість покращення через інверсію сегментів туру:

```
Для кожної пари ребер (i,i+1) та (j,j+1) в турі:
  Якщо d[i,j] + d[i+1,j+1] < d[i,i+1] + d[j,j+1]:
    Інвертувати сегмент між i+1 та j
    (покращення знайдено!)
```

Переваги:

1. Гарантоване покращення: Кожна зміна зменшує довжину туру

2. Швидка локальна збіжність: За $O(n^2)$ часу досягається локальний оптимум

3. Детерміністичність: Завжди знаходить покращення якщо воно існує

Слабкість:

- Застрягання в локальних оптимумах: 2-opt локальний оптимум може бути далеким від глобального

Завдання для самостійної роботи

Теоретичні завдання

Завдання 1: Аналіз теорії схем Завдання: Для бінарної хромосоми довжини l=10 розгляньте схему H = **`1*010*`**.

e) Визначте порядок схеми o(H)

f) Визначте довжину визначення δ(H)

g) Скільки конкретних рядків відповідає цій схемі?

h) Якщо pc=0.7 та pm=0.01, обчисліть ймовірність руйнування схеми за одне покоління

i) Якщо схема має середню пристосованість f(H)=80, а середня пристосованість популяції favg=60, та в поколінні t є 15 екземплярів схеми, оцініть очікувану кількість екземплярів у поколінні t+1

Завдання 2: Кодування Грея Завдання:

a) Перетворіть бінарне число **10110** у код Грея
b) Перетворіть код Грея **11011** у стандартне бінарне число
c) Поясніть, чому для оптимізації функції $f(x) = x^2$ на [0, 31] код Грея кращий за стандартне бінарне кодування
d) Створіть таблицю відповідностей для 4-бітних чисел (0-15) у стандартному бінарному коді та коді Грея

Завдання 3: Аналіз збіжності Завдання: Генетичний алгоритм має популяцію N=50 і працює на функції з 100 локальними мінімумами однакової глибини та 1 глобальним мінімумом.

j) Оцініть ймовірність, що початкова випадкова популяція містить хоча б одну особину в басейні притягання глобального мінімуму

k) Якщо використовується елітизм ne=1, яка нижня межа для ймовірності успішного знаходження глобального мінімуму за нескінченний час?

l) Як вплине збільшення розміру популяції до N=200 на ймовірність успіху?

Завдання 4: Порівняння операторів селекції Завдання: Дана популяція з 5 особин з пристосованостями: f = [10, 20, 30, 40, 50].

m) Обчисліть ймовірності вибору кожної особини для:

- Пропорційної селекції
- Ранг-селекції (лінійної)
- Турнірної селекції з k=2 (аналітично)

n) Проведіть 1000 симуляцій кожного методу і порівняйте теоретичні та емпіричні ймовірності

o) Який метод має найсильніший селекційний тиск? Обґрунтуйте математично.

Завдання 5: Неявний паралелізм Завдання:

a) Доведіть, що бінарний рядок довжини l є екземпляром 2^l різних схем
b) Для популяції розміру N=100 і довжини хромосоми l=20, оцініть кількість різних схем порядку $o \leqslant 3$, які представлені в популяції
c) Поясніть, чому концепція неявного паралелізму робить генетичні алгоритми ефективними для великих просторів пошуку

Дослідницькі завдання (підвищеної складності)

Завдання 1: Острівна модель Завдання: Реалізуйте паралельний ГА з острівною моделлю:

p) Створіть 4 “острови” (незалежні популяції по 25 особин)

q) Кожні 10 поколінь мігрує 2 найкращі особини між сусідніми островами

r) Використовуйте різні параметри на кожному острові (різні pm, pc, оператори)

s) Порівняйте з звичайним ГА (N=100) за:

- Якістю рішень
- Різноманітністю
- Стійкістю до локальних оптимумів

Завдання 2: Коеволюція Завдання: Реалізуйте конкурентну коеволюцію для гри:

t) Гра “Камінь-Ножиці-Папір” з 10 послідовних раундів

u) Дві популяції еволюціонують одночасно

v) Стратегія кожного гравця - послідовність з 10 ходів + правила реакції на попередній хід опонента

w) Пристосованість = кількість перемог проти випадкової вибірки з популяції опонента

x) Проаналізуйте динаміку: чи з’являються складні стратегії? Чи є цикли (динамічна рівновага)?

Завдання 15: Гібридний алгоритм з машинним навчанням Завдання: Створіть гібридний алгоритм, що використовує ML для прискорення:

y) Базова задача: оптимізація 10D функції Растрігіна

z) Після перших 20 поколінь навчіть neural network як метамодель функції (використайте всі обчислені точки)

aa) Наступні 30 поколінь працюйте з метамоделлю (швидко)

bb) Кожні 10 поколінь оцінюйте top-5 особин реальною функцією і доучуйте метамодель

cc)Останні 20 поколінь - тільки реальна функція для точності

Порівняйте: - Загальна кількість обчислень реальної функції - Якість фінального рішення - Загальний час роботи

Теми лабораторних робіт

Базові лабораторні роботи (обов'язкові)

Лабораторна робота №1: Основи генетичних алгоритмів Тривалість: 4 години

Мета: Вивчити базові поняття та структуру генетичного алгоритму, реалізувати простий ГА.

Теоретична частина:

- Історія та біологічна аналогія

- Базова структура ГА

- Оператори: селекція, кросинговер, мутація

Практична частина:

1. Реалізувати базовий ГА для одновимірної функції

2. Функція: $f(x) = x \cdot \sin(10\pi x) + 1$, $x \in [0,1]$

3. Бінарне кодування (10 біт)

4. Турнірна селекція (k=2)

5. Одноточковий кросовер

6. Бітова мутація

Завдання:

- Реалізувати всі компоненти ГА

- Знайти глобальний максимум

- Побудувати графік збіжності

- Візуалізувати еволюцію популяції

Звітність: Програмний код, графіки, аналіз (5-7 сторінок)

Лабораторна робота №2: Кодування та представлення рішень

Мета: Вивчити різні методи кодування, порівняти бінарне кодування та код Грея.

Теоретична частина:

- Типи кодування: бінарне, дійсне, перестановочне

- Код Грея: властивості та переваги

- Декодування та нормалізація

Практична частина:

1. Реалізувати дві версії ГА:

- З стандартним бінарним кодуванням

- З кодом Грея

8. Тестова функція: Растрігіна 2D

9. Порівняти:

   - Швидкість збіжності
   - Якість рішень
   - Кількість застрягань у локальних мінімумах

Завдання:

- Створити таблицю відповідностей бінар-Грей для 5 біт

- Реалізувати функції перетворення

- Провести 30 запусків кожного варіанту

- Статистичний аналіз результатів

Звітність: Програмний код, порівняльні таблиці, статистичний аналіз (7-10 сторінок)

Лабораторна робота №3: Оператори селекції

Мета: Дослідити вплив різних методів селекції на ефективність ГА.

Теоретична частина:

- Пропорційна селекція

- Ранг-селекція

- Турнірна селекція

- Селекційний тиск та його вплив

Практична частина: Реалізувати 4 методи селекції:

1. Рулетна (пропорційна)

2. Ранк-селекція

3. Турнірна (k=2)

4. Турнірна (k=5)

Тестова функція: Растрігіна 5D

Завдання:

- Для кожного методу: 30 запусків

- Виміряти: середню якість, std, час збіжності, відсоток успіхів

- Дослідити передчасну збіжність (виміряти різноманітність популяції)

Звітність: Програмний код, порівняльний аналіз, рекомендації щодо вибору методу (8-10 сторінок)

Лабораторна робота №4: Оператори кросинговеру та мутації

Мета: Порівняти різні генетичні оператори, дослідити вплив параметрів.

Теоретична частина:

- Типи кросинговеру: одноточковий, двоточковий, рівномірний, арифметичний

- Типи мутації: бітова, гаусівська, рівномірна

- Ймовірності операторів

Практична частина:

1. Реалізувати для бінарного кодування:

- 3 типи кросинговеру

- Бітову мутацію

10. Реалізувати для дійсного кодування:

   – Арифметичний та SBX кросинговер
   – Гаусівську та полінноміальну мутацію

11. Тестові функції:

   – Растрігіна (багатомодальна)
   – Сфери (унімодальна)
   – Розенброка (долина)

Завдання:

- Дослідити вплив pc (0.5, 0.7, 0.9)

- Дослідити вплив pm (0.01, 0.05, 0.1, 0.2)

- Знайти оптимальні комбінації для кожної функції

- Побудувати поверхні відгуку (якість vs pc vs pm)

Звітність: Програмний код, результати експериментів, 3D-графіки, висновки (10-12 сторінок)

Лабораторна робота №5: Параметри популяції та збіжність

Мета: Дослідити вплив розміру популяції, елітизму та критеріїв зупинки.

Теоретична частина:

- Розмір популяції та його вплив

- Елітизм та монотонність збіжності

- Критерії зупинки: стагнація, різноманітність

Практична частина:

1. Дослідити вплив розміру популяції: - N = 20, 50, 100, 200, 500
12. Дослідити елітизм:
    - Без елітизму
    - ne = 1, 2, 5, 10
13. Реалізувати критерії зупинки:
    - Стагнація (k=20 поколінь, ε=0.01)
    - Різноманітність (порогове значення)

Завдання:

- Побудувати графіки: якість - N, час - N

- Перевірити теорему збіжності з елітизмом

- Порівняти різні критерії зупинки

- Знайти оптимальний баланс N - якість - час

Звітність: Програмний код, експериментальні дані, графіки, рекомендації (8-10 сторінок)

Спеціалізовані лабораторні роботи (за вибором)

Лабораторна робота №6: Генетичний алгоритм для задачі комівояжера

Мета: Розробити ефективний ГА для задачі комівояжера з спеціалізованими операторами.

Теоретична частина:

- Задача комівояжера: формулювання, складність

- Перестановочне кодування

- Спеціалізовані оператори: OX, PMX, Edge Recombination

- Мутації: swap, insert, inversion

Практична частина:

1. Згенерувати тестові набори: - 30 міст (легкий) - 50 міст (середній) - 100 міст (складний)

14. Реалізувати оператори:

    - Order Crossover (OX)
    - Partially Mapped Crossover (PMX)
    - Swap та Inversion мутації

15. Реалізувати 2-opt локальний пошук

Завдання:

- Порівняти OX vs PMX

- Додати 2-opt та оцінити покращення

- Порівняти з жадібним алгоритмом

- Візуалізувати найкращі знайдені тури

- Дослідити масштабованість (час - кількість міст)

Звітність: Програмний код, візуалізації турів, порівняльний аналіз, графіки масштабованості (12-15 сторінок)

Лабораторна робота №7: Оптимізація нейронних мереж

Мета: Використати ГА для підбору гіперпараметрів нейронної мережі.

Теоретична частина:

- Гіперпараметри нейронних мереж

- Змішане кодування

- AutoML та нейроеволюція

Практична частина: 1. Задача: класифікація MNIST (спрощена версія)

2. Оптимізовані гіперпараметри:

    - Кількість прихованих шарів (1-3)
    - Нейрони в шарах (32-256)
    - Learning rate (0.0001-0.1, log scale)

- Dropout (0-0.5)
- Optimizer (SGD/Adam/RMSprop)
- Batch size (32/64/128)

3. Функція пристосованості:

- Точність
- Штраф за складність (кількість параметрів)

Завдання:

- Реалізувати ГА з змішаним кодуванням

- Знайти оптимальну конфігурацію

- Порівняти з випадковим пошуком і базовою мережею - Проаналізувати знайдені архітектури

Звітність: Програмний код, найкращі конфігурації, порівняння методів, аналіз (15-20 сторінок)

Лабораторна робота №8: Складання розкладу занять

Мета: Розробити ГА для практичної задачі складання університетського розкладу.

Теоретична частина:

- Задача складання розкладу: обмеження та цілі

- Жорсткі та м’які обмеження

- Функція пристосованості зі штрафами

Практична частина:

1. Спрощена модель: - 5 груп студентів - 10 викладачів - 8 предметів - 5 аудиторій - 5 днів × 4 пари

2. Обмеження:

- Жорсткі: конфлікти викладачів, груп, аудиторій
- М’які: вікна в розкладі, навантаження викладачів

3. Кодування: матриця призначень

Завдання: - Реалізувати функцію пристосованості з штрафами - Реалізувати repair operator - Знайти допустимий розклад - Мінімізувати порушення м’яких обмежень - Порівняти з жадібним конструктивним алгоритмом

Звітність: Програмний код, приклади розкладів, статистика порушень, порівняльний аналіз (12-15 сторінок)

Лабораторна робота №9: Паралельні та острівні генетичні алгоритми

Мета: Реалізувати та дослідити паралельні архітектури ГА.

Теоретична частина:

- Острівна модель (island model)

- Міграція особин

- Дифузна модель

- Паралелізація обчислень

Практична частина:

1. Реалізувати острівну модель: - 4 острови по 25 особин - Міграція топ-2 кожні 10 поколінь - Різні параметри на островах
2. Топології міграції:
   - Кільце (1→2→3→4→1)
   - Повнозв’язна (кожен з кожним)
   - Зірка (центральний острів)
3. Тестові функції: багатомодальні високої розмірності

Завдання:

- Порівняти з послідовним ГА (N=100)

- Дослідити вплив частоти міграції

- Порівняти топології

- Виміряти прискорення (якщо є паралелізація)

- Проаналізувати різноманітність на островах

Звітність: Програмний код, результати для різних топологій, аналіз ефективності, рекомендації (15-18 сторінок)

Список використаної літератури.

.NET Collections. (2023). https://docs.microsoft.com/en-us/dotnet/standard/collections/

Aaronson, S. (2013). Quantum Computing Since Democritus. Cambridge University Press.

Abdelraheem, M. A., Beelen, P., Bogdanov, A., & Tischhauser, E. (2015). Twisted polynomials and forgery attacks on GCM. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 762-786). Springer.

Abdi, H., & Williams, L. J. (2010). Principal component analysis. Wiley Interdisciplinary Reviews: Computational Statistics, 2(4), 433-459.

Abdul Bari Algorithms. (2023). https://www.youtube.com/channel/UCZCFT11CWBi3MHNlGf019nw

Academia.edu. (2023). https://www.academia.edu/

ACM Digital Library. (2023). https://dl.acm.org/

Adelson-Velsky, G., & Landis, E. M. (1962). An algorithm for organization of information. Proceedings of the USSR Academy of Sciences, 146, 263-266.

Aggarwal, A., & Vitter, J. S. (1988). “The input/output complexity of sorting and related problems”. Communications of the ACM, 31(9), 1116-1127.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). The Design and Analysis of Computer Algorithms. Addison-Wesley.

Akenine-Möller, T., Haines, E., & Hoffman, N. (2018). Real-Time Rendering (4th ed.). A K Peters/CRC Press.

Akl, S. G. (1985). Parallel Sorting Algorithms. Academic Press.

AlgoExpert. (2023). Algorithm Interview Preparation. https://www.algoexpert.io/

AlgoMonster. (2023). Algorithm Pattern Recognition. https://algo.monster/

Algorithm Animations. (2023). https://www.toptal.com/developers/sorting-algorithms

Algorithm Visualizer. (2023). https://algorithm-visualizer.org/

Algorithm Wiki. (2023). https://algorithm-wiki.org/

Algorithms Live!. (2023). https://www.youtube.com/channel/UCBLr7ISa_YDy5qeATupf26w

Alman, J., & Williams, V. V. (2021). A refined laser method and faster matrix multiplication. In Proceedings of SODA 2021 (pp. 522-539).

Anderson, C., Gibbons, P., & Matias, Y. (2002). “Fast incremental maintenance of approximate histograms”. ACM Transactions on Database Systems, 27(3), 261-298.

Ankerst, M., Breunig, M. M., Kriegel, H. P., & Sander, J. (1999). “OPTICS: Ordering points to identify the clustering structure”. ACM SIGMOD Record, 28(2), 49-60.

Anthropic (2023). Constitutional AI: Harmlessness from AI feedback. arXiv preprint.

Arge, L., Bender, M. A., Demaine, E. D., Holland-Minkley, B., & Munro, J. I. (2007). “An optimal cache-oblivious priority queue and its application to graph algorithms”. SIAM Journal on Computing, 36(6), 1672-1695.

Arora, S., Barak, B. “Computational Complexity: A Modern Approach”. Cambridge University Press, 2009.

Arthur, D., & Vassilvitskii, S. (2007). “k-means++: The advantages of careful seeding”. Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 1027-1035.

ArXiv.org. (2023). Computer Science - Data Structures and Algorithms. https://arxiv.org/list/cs.DS/recent

Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., & Wu, A. Y. (1998). “An optimal algorithm for approximate nearest neighbor searching in fixed dimensions”. Journal of the ACM, 45(6), 891-923.

AtCoder. (2023). Japanese Competitive Programming Platform. https://atcoder.jp/

Atkin, A. O. L., & Bernstein, D. J. (2004). Prime sieves using binary quadratic forms. Mathematics of Computation, 73(246), 1023-1030.

Aumasson, J. P., Henzen, L., Meier, W., & Phan, R. C. W. (2014). The hash function BLAKE. Springer.

Aumasson, J. P., Neves, S., Wilcox-O'Hearn, Z., & Winnerlein, C. (2013). BLAKE2: simpler, smaller, fast as MD5. In Applied Cryptography and Network Security (pp. 119-135). Springer.

Axtmann, M., Bingmann, T., Sanders, P., & Schulz, C. (2015). "Practical massively parallel sorting". Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures.

Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). "Models and issues in data stream systems". Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 1-16.

Bach, E., & Shallit, J. (1996). Algorithmic Number Theory, Volume 1: Efficient Algorithms. MIT Press. Chapter 5: Chinese Remainder Theorem.

Bach, E., & Shallit, J. (1996). Algorithmic Number Theory, Volume 1: Efficient Algorithms. MIT Press. Chapter 8: Prime Number Sieves.

Bach, E., & Shallit, J. (1996). Algorithmic Number Theory, Volume 1: Efficient Algorithms. MIT Press. Chapter 9: Probabilistic Primality Tests.

Bach, E., & Shallit, J. (1996). Algorithmic Number Theory, Volume 1: Efficient Algorithms. MIT Press. Section 8.4: The Sieve of Eratosthenes.

Bach, E., & Shallit, J. (1996). Algorithmic Number Theory, Volume 1: Efficient Algorithms. MIT Press.

Back to Back SWE. (2023). Data Structures Explanations. https://backtobackswe.com/

Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., & Stoica, I. (2003). "Looking up data in P2P systems". Communications of the ACM, 46(2), 43-48.

Ballard, G., Demmel, J., Holtz, O., & Schwartz, O. (2011). Minimizing communication in numerical linear algebra. SIAM Journal on Matrix Analysis and Applications, 32(3), 866-901.

Bandi, N., Agrawal, D., & El Abbadi, A. (2007). "Fast algorithms for heavy distinct hitters using associative memories". 27th International Conference on Distributed Computing Systems, 6-6.

Barber, C. B., Dobkin, D. P., & Huhdanpaa, H. (1996). "The Quickhull algorithm for convex hulls". ACM Transactions on Mathematical Software, 22(4), 469-483.

Base.cs Podcast. (2023). https://www.codenewbie.org/basecs

Batcher, K. E. (1968). "Sorting networks and their applications". Proceedings of the April 30–May 2, 1968, spring joint computer conference. ACM. pp. 307–314.

Baumgart, I., & Mies, S. (2007). “S/Kademlia: A practicable approach towards secure key-based routing”. International Conference on Parallel and Distributed Systems, 1-8.

Bayer, R., & McCreight, E. (1972). Organization and maintenance of large ordered indices. Acta Informatica, 1(3), 173-189.

Beckmann, N., Kriegel, H. P., Schneider, R., & Seeger, B. (1990). “The R-tree: An Efficient and Robust Access Method for Points and Rectangles”. Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 322-331.

Belkina, A. C., et al. (2019). Automated optimized parameters for t-distributed stochastic neighbor embedding improve visualization and analysis of large datasets. Nature Communications, 10(1), 5415.

Bellman, R. (1958). On a routing problem. Quarterly of Applied Mathematics, 16(1), 87-90.

Bender, M. A. (2023). Stony Brook Algorithm Repository. http://www3.cs.stonybrook.edu/~algorith/

Bender, M. A., Demaine, E. D., & Farach-Colton, M. (2005). “Cache-oblivious B-trees”. SIAM Journal on Computing, 35(2), 341-358.

Benet, J. (2014). “IPFS - Content Addressed, Versioned, P2P File System”. arXiv preprint arXiv:1407.3561.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2), 157-166.

Bennett, K. P., & Campbell, C. (2000). Support vector machines: hype or hallelujah?. ACM SIGKDD Explorations Newsletter, 2(2), 1-13.

Bentley, J. (2000). Programming Pearls (2nd ed.). Addison-Wesley.

Bentley, J. L. (1975). “Multidimensional binary search trees used for associative searching”. Communications of the ACM, 18(9), 509-517.

Bentley, J. L., & Ottmann, T. A. (1979). “Algorithms for reporting and counting geometric intersections”. IEEE Transactions on Computers, C-28(9), 643-647.

Bentley, J. L., & Shamos, M. I. (1976). “Divide-and-conquer in multidimensional space”. Proceedings of the eighth annual ACM symposium on Theory of computing, 220-230.

Berinde, R., Cormode, G., Indyk, P., & Strauss, M. J. (2009). “Space-optimal heavy hitters with strong error bounds”. Proceedings of the 28th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 157-166.

Bernstein, D. J. (2004). The Sieve of Atkin. Available at: https://cr.yp.to/papers/primesieves.pdf

Bernstein, D. J. (2008). ChaCha, a variant of Salsa20. Workshop Record of SASC, 8(1), 3-5.

Bernstein, D. J., & Lange, T. (2007). Faster addition and doubling on elliptic curves. In Advances in Cryptology–ASIACRYPT 2007 (pp. 29-50). Springer.

Bernstein, D. J., & Lange, T. (2017). Montgomery curves and the Montgomery ladder. In Topics in Computational Number Theory Inspired by Peter L. Montgomery (pp. 82-115).

Bhargava, A. (2016). Grokking Algorithms: An Illustrated Guide for Programmers. Manning Publications.

Big-O Cheat Sheet. (2023). https://www.bigocheatsheet.com/

Biham, E., & Dunkelman, O. (2007). A framework for iterative hash functions: HAIFA. In Second NIST Cryptographic Hash Workshop.

Bini, D., Capovani, M., Romani, F., & Lotti, G. (1979). O(n^2.7799) complexity for n×n approximate matrix multiplication. Information Processing Letters, 8(5), 234-235.

Bishop, C. M. (2006). “Pattern Recognition and Machine Learning”. Springer.

Blake, I., Seroussi, G., & Smart, N. (1999). Elliptic Curves in Cryptography. Cambridge University Press.

BLAKE2 official website. (2023). BLAKE2 - fast secure hashing. Retrieved from https://blake2.net/

Blelloch, G. E. (1990). “Prefix sums and their applications”. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.

Blelloch, G. E., Gibbons, P. B., & Simhadri, H. V. (2008). “Low depth cache-oblivious algorithms”. Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures, 189-199.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), 422-426.

Bluestein, L. (1970). A linear filtering approach to the computation of discrete Fourier transform. IEEE Transactions on Audio and Electroacoustics, 18(4), 451-455.

Blum, A. (2023). Machine Learning Theory. https://www.cs.cmu.edu/~avrim/

Böck, H., Zauner, A., Devlin, S., Somorovsky, J., & Jovanovic, P. (2016). Nonce-disrespecting adversaries: Practical forgery attacks on GCM in TLS. In 10th USENIX Workshop on Offensive Technologies.

Bommasani, R., et al. (2021). On the opportunities and risks of foundation models. arXiv preprint.

Boneh, D. (1999). Twenty years of attacks on the RSA cryptosystem. Notices of the AMS, 46(2), 203-213.

Boneh, D., & Shoup, V. (2020). A Graduate Course in Applied Cryptography. Retrieved from https://crypto.stanford.edu/~dabo/cryptobook/

Boost Graph Library. (2023). C++ Graph Algorithms. https://www.boost.org/doc/libs/release/libs/graph/

Bos, J. W., & Coster, M. J. (1990). Addition chain heuristics. In Advances in Cryptology—CRYPTO'89 Proceedings (pp. 400-407). Springer.

Brass, P. (2008). Advanced Data Structures. Cambridge University Press.

Brauer, A. (1939). On addition chains. Bulletin of the American Mathematical Society, 45(10), 736-739.

Breiman, L. (2001). Random forests. Machine learning, 45(1), 5-32.

Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. A. (1984). Classification and regression trees. CRC press.

Bressoud, D. M. (1989). Factorization and Primality Testing. Springer-Verlag. Chapter 5: Sieve Methods.

Brigham, E. O. (1988). The Fast Fourier Transform and Its Applications. Prentice Hall.

Brilliant.org. (2023). Computer Science Fundamentals. https://brilliant.org/courses/computer-science/

Brodal, G. S., & Fagerberg, R. (2003). "Cache oblivious distribution sweeping". Proceedings of the 29th International Colloquium on Automata, Languages and Programming, 426-438.

Brown, T., et al. (2020). Language models are few-shot learners. NeurIPS.

Brown, T., Mann, B., Ryder, N., et al. (2020). Language models are few-shot learners. NeurIPS.

Bryant, A., & Cios, K. (2018). “RNN-DBSCAN: A density-based clustering algorithm using reverse nearest neighbor density estimates”. IEEE Transactions on Knowledge and Data Engineering, 30(6), 1109-1121.

B-Tree Visualization. (2023). https://www.cs.usfca.edu/~galles/visualization/BTree.html

Bubeck, S., et al. (2023). Sparks of artificial general intelligence: Early experiments with GPT-4. arXiv preprint.

Bühlmann, P., & Hothorn, T. (2007). Boosting algorithms: Regularization, prediction and model fitting. Statistical Science, 22(4), 477-505.

Burrows, M. (2006). “The Chubby lock service for loosely-coupled distributed systems”. Proceedings of the 7th symposium on Operating systems design and implementation, 335-350.

C++ Reference. (2023). https://en.cppreference.com/w/cpp/algorithm

Campello, R. J., Moulavi, D., & Sander, J. (2013). “Density-based clustering based on hierarchical density estimates”. Pacific-Asia Conference on Knowledge Discovery and Data Mining, 160-172.

Candès, E. J., Li, X., Ma, Y., & Wright, J. (2011). Robust principal component analysis? Journal of the ACM, 58(3), 1-37.

Cao, F., Estert, M., Qian, W., & Zhou, A. (2006). “Density-based clustering over an evolving data stream with noise”. Proceedings of the 2006 SIAM International Conference on Data Mining, 328-339.

Capannini, G., Silvestri, F., & Baraglia, R. (2012). “Sorting on GPUs for large scale datasets: A thorough comparison”. Information Processing & Management, 48(5), 903-917.

Carnegie Mellon University. (2023). 15-451/651: Algorithm Design and Analysis. https://www.cs.cmu.edu/~15451/

Carnegie Mellon University. (2023). 15-451/651: Algorithm Design and Analysis. https://www.cs.cmu.edu/~15451/

Chabchoub, Y., & Hebrail, G. (2010). “Sliding HyperLogLog: Estimating cardinality in a data stream over a sliding window”. Data Mining Workshops (ICDMW), 1297-1303.

Chandra, T. D., Griesemer, R., & Redstone, J. (2007). “Paxos made live: an engineering perspective”. Proceedings of the 26th annual ACM symposium on Principles of distributed computing, 398-407.

Chang, C. C., & Lin, C. J. (2011). LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2(3), 1-27.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2008). “Bigtable: A distributed storage system for structured data”. ACM Transactions on Computer Systems, 26(2), 1-26.

Charikar, M., Chen, K., & Farach-Colton, M. (2002). “Finding frequent items in data streams”. International Colloquium on Automata, Languages, and Programming, 693-703.

Chassaing, P., & Gerin, L. (2006). “Efficient estimation of the cardinality of large data sets”. Proceedings of the 4th Colloquium on Mathematics and Computer Science, 419-422.

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining (pp. 785-794).

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining (pp. 785-794).

Chen, Y., Zhou, L., Bouguila, N., Wang, C., Chen, Y., & Du, J. (2021). “BLOCK-DBSCAN: Fast clustering for large scale data”. Pattern Recognition, 109, 107624.

Cho, K., et al. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. EMNLP.

CiteSeerX. (2023). http://citeseerx.ist.psu.edu/

Cockroach Labs. (2020). “CockroachDB: The Resilient SQL Database”. Technical documentation on Raft implementation.

Codecademy Pro. (2023). Advanced Algorithms Track. https://www.codecademy.com/learn/paths/advanced-algorithms

Codecademy. (2023). Computer Science Path. https://www.codecademy.com/learn/paths/computer-science

CodeChef. (2023). Competitive Programming Platform. https://www.codechef.com/

Codeforces. (2023). Competitive Programming and Algorithms. https://codeforces.com/

Coding Interview University. (2023). https://github.com/jwasham/coding-interview-university

Cohen, H. (1993). A Course in Computational Algebraic Number Theory. Graduate Texts in Mathematics. Springer-Verlag.

Cohen, H. (1993). A Course in Computational Algebraic Number Theory. Springer-Verlag.

Cohen, H., Frey, G., et al. (2005). Handbook of Elliptic and Hyperelliptic Curve Cryptography. Chapman & Hall/CRC.

Cole, R. (1988). “Parallel merge sort”. SIAM Journal on Computing, 17(4), 770-785.

Collins, M. (2002). Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. EMNLP.

Competitive Programming 4. (2023). CP4 Book Website. https://cpbook.net/

Complexity Zoo. (2023). https://complexityzoo.net/

Cook, S.A. “The Complexity of Theorem-Proving Procedures”. Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, 1971.

Cooley, J. W., & Tukey, J. W. (1965). An algorithm for the machine calculation of complex Fourier series. Mathematics of computation, 19(90), 297-301.

Coppersmith, D., & Winograd, S. (1990). Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation, 9(3), 251-280.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). “Introduction to Algorithms” (3rd ed.). MIT Press. Chapter 27: Multithreaded Algorithms.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). “Introduction to Algorithms” (3rd ed.). MIT Press. Chapter 33.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). “Introduction to Algorithms” (3rd ed.). MIT Press. Chapter 33: Computational Geometry. - Класичний підручник з алгоритмів з детальним описом алгоритму сканування Грема.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). “Introduction to Algorithms” (3rd ed.). MIT Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press. Chapter 31.5: The Chinese remainder theorem.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press. Chapter 31.8: Primality Testing.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press. Chapter 31: Number-Theoretic Algorithms.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press. Exercise 31.2-5: The Sieve of Eratosthenes.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Cormode, G. (2011). “Sketch techniques for approximate query processing”. Foundations and Trends in Databases, 3(3-4), 182-234.

Cormode, G., & Hadjieleftheriou, M. (2008). “Finding frequent items in data streams”. Proceedings of the VLDB Endowment, 1(2), 1530-1541.

Cormode, G., & Muthukrishnan, S. (2004). “What’s hot and what’s not: tracking most frequent items dynamically”. Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 296-306.

Cormode, G., & Muthukrishnan, S. (2005). “An improved data stream summary: the count-min sketch and its applications”. Journal of Algorithms, 55(1), 58-75.

Cosmos. (2023). World’s Largest Algorithmic Library. https://github.com/OpenGenus/cosmos

Coursera. (2023). Data Structures and Algorithms Specialization. https://www.coursera.org/specializations/data-structures-algorithms

Cox, D. R. (1958). “The regression analysis of binary sequences”. Journal of the Royal Statistical Society, Series B, 20(2), 215-242.

CP-Algorithms. (2023). Algorithms for Competitive Programming. https://cp-algorithms.com/

Crandall, R., & Pomerance, C. (2005). Prime Numbers: A Computational Perspective (2nd ed.). Springer. Chapter 3.6: The Sieve of Atkin.

Crandall, R., & Pomerance, C. (2005). Prime Numbers: A Computational Perspective (2nd ed.). Springer. Chapter 3: Recognizing Primes and Composites.

Crandall, R., & Pomerance, C. (2005). Prime Numbers: A Computational Perspective (2nd ed.). Springer. Chapter 4: Primality Proving.

Crandall, R., & Pomerance, C. (2005). Prime Numbers: A Computational Perspective (2nd ed.). Springer. Section 2.1.3: Chinese Remainder Theorem.

Crane, C. A. (1972). Linear lists and priority queues as balanced binary trees. Stanford University Technical Report.

Cristianini, N., & Shawe-Taylor, J. (2000). An Introduction to Support Vector Machines and Other Kernel-based Learning Methods. Cambridge University Press.

Cristianini, N., & Shawe-Taylor, J. (2000). An Introduction to Support Vector Machines. Cambridge University Press.

Crosby, S. A., & Wallach, D. S. (2007). “An analysis of BitTorrent’s two Kademlia-based DHTs”. Technical Report TR-07-04, Department of Computer Science, Rice University.

CS Dojo. (2023). https://www.youtube.com/channel/UCxX9wt5FWQUAAz4UrysqK9A

Dabek, F., Brunskill, E., Kaashoek, M. F., Karger, D., Morris, R., Stoica, I., & Balakrishnan, H. (2001). “Building peer-to-peer systems with Chord, a distributed lookup service”. Proceedings of the 8th Workshop on Hot Topics in Operating Systems.

Daily Coding Problem. (2023). https://www.dailycodingproblem.com/

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. NeurIPS.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). Algorithms. McGraw-Hill.

Data Structure Visualizations. (2023). https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

DataCamp. (2023). Data Structures and Algorithms in Python. https://www.datacamp.com/courses/data-structures-and-algorithms-in-python

Davis, M., Sigal, R., Weyuker, E.J. “Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science” (2nd edition). Academic Press, 1994.

Dayan, N., Athanassoulis, M., & Idreos, S. (2017). “Monkey: Optimal navigable key-value store”. Proceedings of the 2017 ACM International Conference on Management of Data, 79-94.

DBLP Computer Science Bibliography. (2023). https://dblp.org/

de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). “Computational Geometry: Algorithms and Applications” (3rd ed.). Springer-Verlag. Chapter 1: Convex Hulls. - Фундаментальний підручник з обчислювальної геометрії.

de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). “Computational Geometry: Algorithms and Applications” (3rd ed.). Springer-Verlag. Chapter 2.

de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). “Computational Geometry: Algorithms and Applications” (3rd ed.). Springer-Verlag.

Dean, J., & Ghemawat, S. (2004). “MapReduce: Simplified Data Processing on Large Clusters”. Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, OSDI’04.

Dean, J., & Ghemawat, S. (2008). “MapReduce: simplified data processing on large clusters”. Communications of the ACM, 51(1), 107-113.

Dechev, D., Pirkelbauer, P., & Stroustrup, B. (2010). “Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs”. 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 185-192.

Defays, D. (1977). “An efficient algorithm for a complete link method”. The Computer Journal, 20(4), 364-366.

Demaine, E. (2023). Advanced Data Structures Course. http://courses.csail.mit.edu/6.851/

Demaine, E. D. (2002). “Cache-oblivious algorithms and data structures”. Lecture Notes from the EEF Summer School on Massive Data Sets, 1-29.

Demaine, E. D., & Iacono, J. (2021). “A survey of cache-oblivious data structures”. Algorithms and Data Structures: The Basic Toolbox, 389-412.

Dev.to Algorithms Tag. (2023). https://dev.to/t/algorithms

Developer Tea. (2023). https://spec.fm/podcasts/developer-tea

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Dictionary of Algorithms and Data Structures. (2023). https://xlinux.nist.gov/dads/

Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE Transactions on Information Theory, 22(6), 644-654.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269-271.

Dinic, E. A. (1970). Algorithm for solution of a problem of maximum flow in networks with power estimation. Soviet Mathematics Doklady, 11, 1277-1280.

Discord Algorithm Servers. (2023). https://discord.gg/algorithms

Domingos, P., & Pazzani, M. (1997). On the optimality of the simple Bayesian classifier under zero-one loss. Machine learning, 29(2), 103-130.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., et al. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. ICLR.

Duhamel, P., & Vetterli, M. (1990). Fast Fourier transforms: a tutorial review and a state of the art. Signal processing, 19(4), 259-299.

Durand, M., & Flajolet, P. (2003). “Loglog counting of large cardinalities”. In European Symposium on Algorithms, 605-617.

Dutré, P., Bala, K., & Bekaert, P. (2006). Advanced Global Illumination (2nd ed.). A K Peters.

Dworkin, M. (2001). Recommendation for block cipher modes of operation: Methods and techniques. NIST special publication, 800-38A.

Eddy, W. F. (1977). “A new convex hull algorithm for planar sets”. ACM Transactions on Mathematical Software, 3(4), 398-403.

Edmonds, J. (1965). Paths, trees, and flowers. Canadian Journal of Mathematics, 17, 449-467.

Edmonds, J., & Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM, 19(2), 248-264.

Educative.io. (2023). Grokking the Coding Interview. https://www.educative.io/courses/grokking-the-coding-interview

edX. (2023). Algorithmic Design and Techniques. https://www.edx.org/learn/algorithms

Efron, B., & Hastie, T. (2016). “Computer Age Statistical Inference: Algorithms, Evidence, and Data Science”. Cambridge University Press.

Einziger, G., & Friedman, R. (2014). “TinyLFU: A highly efficient cache admission policy”. 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 146-153.

Elman, J. L. (1990). Finding structure in time. Cognitive Science, 14(2), 179-211.

E-Maxx Algorithms in English. (2023). https://e-maxx-eng.appspot.com/

Eppstein, D. (2023). Data Structures Blog. https://11011110.github.io/blog/

Erickson, J. (2019). Algorithms. University of Illinois. http://jeffe.cs.illinois.edu/teaching/algorithms/

Erickson, J. (2023). Algorithms Course Materials. http://jeffe.cs.illinois.edu/teaching/algorithms/

Errichto. (2023). https://www.youtube.com/channel/UCBr_Fu6q9iHYQCh13jmpbrg

Ertl, O. (2017). “New cardinality estimation algorithms for HyperLogLog sketches”. arXiv preprint arXiv:1702.01284.

Ester, M., Kriegel, H. P., Sander, J., & Xu, X. (1996). “A density-based algorithm for discovering clusters in large spatial databases with noise”. Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, 226-231.

Estivill-Castro, V., & Wood, D. (1992). “A survey of adaptive sorting algorithms”. ACM Computing Surveys, 24(4), 441-476.

Facebook (2021). “RocksDB: A Persistent Key-Value Store for Fast Storage Environments”. RocksDB Documentation.

Facebook Hacker Cup. (2023). https://www.facebook.com/codingcompetitions/hacker-cup

Fan, B., Andersen, D. G., Kaminsky, M., & Mitzenmacher, M. D. (2014). Cuckoo filter: Practically better than bloom. In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies (pp. 75-88).

Ferguson, N. (2005). Authentication weaknesses in GCM. Comments to NIST.

Ferguson, N., Schneier, B., & Kohno, T. (2010). Cryptography Engineering: Design Principles and Practical Applications. Wiley.

Flajolet, P., & Martin, G. N. (1985). “Probabilistic counting algorithms for data base applications”. Journal of Computer and System Sciences, 31(2), 182-209.

Flajolet, P., Fusy, É., Gandouet, O., & Meunier, F. (2007). “HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm”. In AofA: Analysis of Algorithms, 137-156.

Floyd, R. W. (1962). Algorithm 97: Shortest path. Communications of the ACM, 5(6), 345.

Ford, L. R., & Fulkerson, D. R. (1956). Maximal flow through a network. Canadian Journal of Mathematics, 8, 399-404.

Formation. (2023). Algorithm Training. https://formation.dev/

Fraser, K. (2004). “Practical lock-freedom”. PhD thesis, University of Cambridge Computer Laboratory.

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM, 34(3), 596-615.

FreeCodeCamp. (2023). Algorithms and Data Structures. https://www.freecodecamp.org/learn/javascript-algorithms-and-data-structures/

Freedman, M. J., & Mazieres, D. (2003). “Sloppy hashing and self-organizing clusters”. International Workshop on Peer-to-Peer Systems.

Freund, Y., & Schapire, R. E. (1999). Large margin classification using the perceptron algorithm. Machine Learning, 37(3), 277-296.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. Annals of statistics, 1189-1232.

Friedman, J. H. (2002). Stochastic gradient boosting. Computational statistics & data analysis, 38(4), 367-378.

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1977). “An algorithm for finding best matches in logarithmic expected time”. ACM Transactions on Mathematical Software, 3(3), 209-226.

Frigo, M., & Johnson, S. G. (2005). The design and implementation of FFTW3. Proceedings of the IEEE, 93(2), 216-231.

Frigo, M., & Strumpen, V. (2009). “The cache complexity of multithreaded cache oblivious algorithms”. Theory of Computing Systems, 45(2), 203-233.

Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). “Cache-oblivious algorithms”. Proceedings of the 40th Annual Symposium on Foundations of Computer Science, 285-297.

Galway, W. F. (2000). Dissecting a sieve to cut its need for space. In Algorithmic Number Theory (pp. 297-312). Springer.

Gan, J., & Tao, Y. (2015). “DBSCAN revisited: Mis-claim, un-fixability, and approximation”. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 519-530.

Gates, A. (2011). “Programming Pig”. O’Reilly Media.

Gathen, J., & Gerhard, J. (2013). Modern Computer Algebra (3rd ed.). Cambridge University Press. Chapter 5.4: Chinese Remainder Algorithm.

GeeksforGeeks. (2023). Data Structures and Algorithms. https://www.geeksforgeeks.org/data-structures/

Gentleman, W. M., & Sande, G. (1966). Fast Fourier transforms: for fun and profit. In Proceedings of the November 7-10, 1966, fall joint computer conference (pp. 563-578).

Gentleman, W. M., & Sande, G. (1966). Fast Fourier transforms: for fun and profit. Proceedings of AFIPS, 29, 563-578.

Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. Machine learning, 63(1), 3-42.

Glassner, A. S. (Ed.). (1989). An Introduction to Ray Tracing. Academic Press.

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. AISTATS.

Go Data Structures. (2023). https://github.com/emirpasic/gods

Go Documentation. (2023). https://golang.org/doc/

Gödel, K. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme”. Monatshefte für Mathematik, 1931.

Goldberg, A. V., & Tarjan, R. E. (1988). A new approach to the maximum-flow problem. Journal of the ACM, 35(4), 921-940.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press. Chapter 10: Sequence Modeling.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

Goodrich, M. T., & Tamassia, R. (2014). Algorithm Design and Applications. Wiley.

Google (2013). “HyperLogLog++: Google’s Take On Engineering HLL”. Google Research Blog.

Google Code Jam Archives. (2023). https://codingcompetitions.withgoogle.com/codejam

Google Scholar. (2023). https://scholar.google.com/

Gordon, D. M. (1998). A survey of fast exponentiation methods. Journal of Algorithms, 27(1), 129-146.

Goyal, A., Daumé III, H., & Cormode, G. (2012). “Sketch algorithms for estimating point queries in NLP”. Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, 1093-1103.

Graefe, G. (2006). “Implementing sorting in database systems”. ACM Computing Surveys, 38(3), 10.

Graham, R. L. (1972). “An efficient algorithm for determining the convex hull of a finite planar set”. Information Processing Letters, 1(4), 132-133. - Оригінальна стаття Рональда Грема, де вперше описано алгоритм.

Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). Introduction to Parallel Computing (2nd ed.). Addison-Wesley.

Graph Online. (2023). https://graphonline.ru/en/

Graves, A. (2013). Generating sequences with recurrent neural networks. arXiv preprint arXiv:1308.0850.

Gray, J., & Lamport, L. (2006). “Consensus on transaction commit”. ACM Transactions on Database Systems, 31(1), 133-160.

Gueron, S. (2013). AES-GCM for efficient authenticated encryption–ending the reign of HMAC-SHA-1?. In Real World Cryptography Workshop.

Gueron, S., & Krasnov, V. (2014). The fragility of AES-GCM authentication algorithm. In 11th International Conference on Information Technology: New Generations (pp. 333-337). IEEE.

Guibas, L. J., & Sedgewick, R. (1978). A dichromatic framework for balanced trees. In Proceedings of the 19th Annual Symposium on Foundations of Computer Science (pp. 8-21).

Guttman, A. (1984). “R-trees: A Dynamic Index Structure for Spatial Searching”. Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 47-57.

HackerRank. (2023). Data Structures Practice Problems. https://www.hackerrank.com/domains/data-structures

Haines, E., & Akenine-Möller, T. (Eds.). (2019). Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs. Apress.

Hamerly, G., & Elkan, C. (2003). “Learning the k in k-means”. Advances in Neural Information Processing Systems, 16, 281-288.

Hankerson, D., Menezes, A., & Vanstone, S. (2004). Guide to Elliptic Curve Cryptography. Springer.

Harris, T., Larus, J., & Rajwar, R. (2010). “Transactional Memory” (2nd Edition). Morgan & Claypool Publishers.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.

Hash Table Visualizer. (2023). https://www.cs.usfca.edu/~galles/visualization/OpenHash.html

Hashicorp. (2020). “Consul: Service Mesh and Service Discovery”. Documentation on Raft consensus in Consul.

Hashnode Algorithms. (2023). https://hashnode.com/n/algorithms

Haskell Data Structures. (2023). https://hackage.haskell.org/package/containers

Hastie, T., Tibshirani, R., & Friedman, J. (2009). “The Elements of Statistical Learning: Data Mining, Inference, and Prediction” (2nd ed.). Springer.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). “The Elements of Statistical Learning” (2nd ed.). Springer.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The elements of statistical learning: data mining, inference, and prediction. Springer Science & Business Media.

Haykin, S. (2009). Neural Networks and Learning Machines (3rd ed.). Pearson.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. CVPR.

Heap Visualizer. (2023). https://heap-visualizer.netlify.app/

Hendler, D., Shavit, N., & Yerushalmi, L. (2004). “A scalable lock-free stack algorithm”. Proceedings of the 16th annual ACM symposium on Parallelism in algorithms and architectures, 206-215.

Herlihy, M., & Shavit, N. (2012). “The Art of Multiprocessor Programming” (Revised Edition). Morgan Kaufmann. Chapter 10: Concurrent Queues and the ABA Problem.

Herlihy, M., & Shavit, N. (2012). “The Art of Multiprocessor Programming” (Revised Edition). Morgan Kaufmann. Chapter 11: Concurrent Stacks and Elimination.

Heule, S., Nunkesser, M., & Hall, A. (2013). “HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm”. Proceedings of the 16th International Conference on Extending Database Technology, 683-692.

Higham, N. J. (1990). Exploiting fast matrix multiplication within the level 3 BLAS. ACM Transactions on Mathematical Software, 16(4), 352-368.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735-1780.

Hoerl, A. E., & Kennard, R. W. (1970). “Ridge regression: Biased estimation for nonorthogonal problems”. Technometrics, 12(1), 55-67.

Hoffmann, J., et al. (2022). Training compute-optimal large language models. NeurIPS.

Hoffstein, J., Pipher, J., & Silverman, J. H. (2008). An Introduction to Mathematical Cryptography. Springer.

Hopcroft, J. E., & Karp, R. M. (1973). An n^5/2 algorithm for maximum matchings in bipartite graphs. SIAM Journal on Computing, 2(4), 225-231.

Hopcroft, J.E., Motwani, R., Ullman, J.D. “Introduction to Automata Theory, Languages, and Computation” (3rd edition). Pearson, 2006.

Hosmer, D. W., Lemeshow, S., & Sturdivant, R. X. (2013). “Applied Logistic Regression” (3rd ed.). Wiley.

Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components. Journal of Educational Psychology, 24(6), 417-441.

Howard, A. G., Zhu, M., Chen, B., et al. (2017). MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.

Howard, H., Malkhi, D., & Spiegelman, A. (2021). “Flexible Paxos: Quorum Intersection Revisited”. arXiv preprint arXiv:1608.06696.

Howard, H., Schwarzkopf, M., Madhavapeddy, A., & Crowcroft, J. (2015). “Raft Refloated: Do We Have Consensus?”. ACM SIGOPS Operating Systems Review, 49(1), 12-21.

Huang, Z. (1998). “Extensions to the k-means algorithm for clustering large data sets with categorical values”. Data Mining and Knowledge Discovery, 2(3), 283-304.

ICPC. (2023). International Collegiate Programming Contest. https://icpc.global/

IEEE Xplore. (2023). https://ieeexplore.ieee.org/

IETF RFC 7693. (2015). The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC). Internet Engineering Task Force.

InterviewBit. (2023). Programming Interview Questions. https://www.interviewbit.com/

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. ICML.

IOI. (2023). International Olympiad in Informatics. https://ioinformatics.org/

Ishwaran, H., & Kogalur, U. B. (2007). Random survival forests for R. R news, 7(2), 25-31.

Iwata, T., Ohashi, K., & Minematsu, K. (2012). Breaking and repairing GCM security proofs. In Advances in Cryptology–CRYPTO 2012 (pp. 31-49). Springer.

Jain, A. K. (2010). “Data clustering: 50 years beyond K-means”. Pattern Recognition Letters, 31(8), 651-666.

Jaja, J. (1992). “An Introduction to Parallel Algorithms”. Addison-Wesley. Chapter 4: Sorting and Selection.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). “An Introduction to Statistical Learning with Applications in R”. Springer.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An introduction to statistical learning. Springer.

Jarvis, R. A. (1973). “On the identification of the convex hull of a finite set of points in the plane”. Information Processing Letters, 2(1), 18-21.

Java API Documentation. (2023). https://docs.oracle.com/javase/8/docs/api/

Java Collections Framework. (2023). https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/package-summary.html

Java Collections Framework. (2023). https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/package-summary.html

JavaScript Algorithms. (2023). https://github.com/trekhleb/javascript-algorithms

JavaScript Algorithms. (2023). https://github.com/trekhleb/javascript-algorithms

Jimenez, R., Osmani, F., & Knutsson, B. (2011). “Sub-second lookups on a large-scale Kademlia-based overlay”. IEEE International Conference on Peer-to-Peer Computing, 82-91.

Jin, C., Qian, W., Sha, C., Yu, J. X., & Zhou, A. (2003). “Dynamically maintaining frequent items over a data stream”. Proceedings of the 12th International Conference on Information and Knowledge Management, 287-294.

Johnson, D. B. (1977). Efficient algorithms for shortest paths in sparse networks. Journal of the ACM, 24(1), 1-13.

Johnson, D., Menezes, A., & Vanstone, S. (2001). The Elliptic Curve Digital Signature Algorithm (ECDSA). International Journal of Information Security, 1(1), 36-63.

Johnson, S. C. (1967). “Hierarchical clustering schemes”. Psychometrika, 32(3), 241-254.

Johnson, S. G., & Frigo, M. (2007). A modified split-radix FFT with fewer arithmetic operations. IEEE Transactions on Signal Processing, 55(1), 111-119.

Jolliffe, I. T. (2002). Principal Component Analysis (2nd ed.). Springer-Verlag.

Joux, A. (2006). Authentication failures in NIST version of GCM. NIST Comment.

JSTOR. (2023). https://www.jstor.org/

Kaggle Learn. (2023). Intro to Programming. https://www.kaggle.com/learn

Kamel, I., & Faloutsos, C. (1994). “Hilbert R-tree: An Improved R-tree using Fractals”. Proceedings of the 20th International Conference on Very Large Data Bases, 500-509.

Kaplan, J., et al. (2020). Scaling laws for neural language models. arXiv preprint.

Karpathy, A., Johnson, J., & Fei-Fei, L. (2015). Visualizing and understanding recurrent networks. arXiv preprint arXiv:1506.02078.

Karumanchi, N. (2016). Data Structures and Algorithms Made Easy. CareerMonk Publications.

Kattis. (2023). Programming Problems. https://open.kattis.com/

Katz, J., & Lindell, Y. (2014). Introduction to Modern Cryptography (2nd ed.). CRC Press.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., ... & Liu, T. Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. Advances in neural information processing systems, 30.

Khan Academy. (2023). Algorithms Course. https://www.khanacademy.org/computing/computer-science/algorithms

Kibriya, A. M., Frank, E., Pfahringer, B., & Holmes, G. (2004). Multinomial naive bayes for text categorization revisited. In Australian Conference on Artificial Intelligence.

King, G., & Zeng, L. (2001). “Logistic regression in rare events data”. Political Analysis, 9(2), 137-163.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Kleinberg, J. (2023). Cornell CS Faculty Page. https://www.cs.cornell.edu/home/kleinber/

Kleinberg, J., & Tardos, E. (2005). “Algorithm Design”. Addison-Wesley.

Kleppmann, M. (2017). “Designing Data-Intensive Applications”. O’Reilly Media. Chapter 9: Consistency and Consensus.

Knuth, D. E. (1997). The Art of Computer Programming, Volume 1: Fundamental Algorithms (3rd ed.). Addison-Wesley.

Knuth, D. E. (1997). The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd ed.). Addison-Wesley. Chapter 4.5.2: The Greatest Common Divisor.

Knuth, D. E. (1998). “The Art of Computer Programming, Volume 3: Sorting and Searching” (2nd ed.). Addison-Wesley.

Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.

Knuth, D. E. (2023). Personal Website. https://www-cs-faculty.stanford.edu/~knuth/

Kobak, D., & Berens, P. (2019). The art of using t-SNE for single-cell transcriptomics. Nature Communications, 10(1), 5416.

Koç, Ç. K. (1995). Analysis of sliding window techniques for exponentiation. Computers & Mathematics with Applications, 30(10), 17-24.

Kotlin Algorithms. (2023). https://github.com/KotlinBy/awesome-kotlin

Kotlin Documentation. (2023). https://kotlinlang.org/docs/home.html

Kozen, D.C. “Theory of Computation”. Springer, 2006.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. NeurIPS.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, 7(1), 48-50.

Kumar, V., Grama, A., Gupta, A., & Karypis, G. (2003). “Introduction to Parallel Computing” (2nd ed.). Addison-Wesley. Chapter 9: Sorting.

Lafore, R. (2017). Data Structures and Algorithms in Java (2nd ed.). Sams Publishing.

Lakshman, A., & Malik, P. (2010). “Cassandra: a decentralized structured storage system”. ACM SIGOPS Operating Systems Review, 44(2), 35-40.

Lam, C. (2010). “Hadoop in Action”. Manning Publications.

Lamport, L. (1998). “The Part-Time Parliament”. ACM Transactions on Computer Systems, 16(2), 133-169.

Lamport, L. (2001). “Paxos Made Simple”. ACM SIGACT News, 32(4), 51-58.

Lamport, L. (2006). “Fast Paxos”. Distributed Computing, 19(2), 79-103.

Lance, G. N., & Williams, W. T. (1967). “A general theory of classificatory sorting strategies: 1. Hierarchical systems”. The Computer Journal, 9(4), 373-380.

Larson, P. Å., & Graefe, G. (1998). “Memory management during run generation in external sorting”. ACM SIGMOD Record, 27(2), 472-483.

Le Gall, F. (2014). Powers of tensors and fast matrix multiplication. In Proceedings of ISSAC 2014 (pp. 296-303).

Lea, D. (2000). “Concurrent Programming in Java: Design Principles and Patterns” (2nd Edition). Addison-Wesley Professional.

Lea, D. (2000). “Concurrent Programming in Java: Design Principles and Patterns” (2nd Edition). Addison-Wesley Professional.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

LeetCode. (2023). Algorithm Problems and Solutions. https://leetcode.com/

Leiserson, C. (2023). MIT CSAIL Page. https://people.csail.mit.edu/cel/

Leutenegger, S. T., Lopez, M. A., & Edgington, J. (1997). “STR: A Simple and Efficient Algorithm for R-tree Packing”. Proceedings of the 13th International Conference on Data Engineering, 497-506.

Lewis, D. D. (1998). Naive (Bayes) at forty: The independence assumption in information retrieval. In European conference on machine learning.

Li, J., Stribling, J., Morris, R., & Kaashoek, M. F. (2005). “Bandwidth-efficient management of DHT routing tables”. Proceedings of the 2nd Symposium on Networked Systems Design and Implementation.

Liaw, A., & Wiener, M. (2002). Classification and regression by randomForest. R news, 2(3), 18-22.

Liberty, E. (2013). “Simple and deterministic matrix sketching”. Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 581-588.

Lin, J., & Dyer, C. (2010). “Data-Intensive Text Processing with MapReduce”. Morgan & Claypool Publishers.

Linderman, G. C., & Steinerberger, S. (2019). Fast interpolation-based t-SNE for improved visualization of single-cell RNA-seq data. Nature Methods, 16(3), 243-245.

Linked List Visualizer. (2023). https://antoniosarosi.github.io/Linked-List-Visualizer/

LinkedIn Learning. (2023). Programming Foundations: Algorithms. https://www.linkedin.com/learning/programming-foundations-algorithms

Liu, Z., Lin, Y., Cao, Y., et al. (2021). Swin transformer: Hierarchical vision transformer using shifted windows. ICCV.

Lloyd, S. (1982). “Least squares quantization in PCM”. IEEE Transactions on Information Theory, 28(2), 129-137.

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. CVPR.

Luo, C., & Carey, M. J. (2019). “LSM-based storage techniques: a survey”. The VLDB Journal, 28(3), 393-418.

Lynch, N., Malkhi, D., & Ratajczak, D. (2002). “Atomic data access in distributed hash tables”. International Workshop on Peer-to-Peer Systems.

MacQueen, J. (1967). “Some methods for classification and analysis of multivariate observations”. Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1, 281-297.

Manku, G. S., & Motwani, R. (2002). “Approximate frequency counts over data streams”. Proceedings of the 28th International Conference on Very Large Data Bases, 346-357.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press.

Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A. N., & Theodoridis, Y. (2006). R-Trees: Theory and Applications. Springer-Verlag London.

Mason, L., Baxter, J., Bartlett, P., & Frean, M. (1999). Boosting algorithms as gradient descent. Advances in neural information processing systems, 12.

Maymounkov, P., & Mazières, D. (2002). “Kademlia: A peer-to-peer information system based on the XOR metric”. International Workshop on Peer-to-Peer Systems, 53-65.

McCallum, A., & Nigam, K. (1998). A comparison of event models for naive bayes text classification. In AAAI-98 workshop on learning for text categorization.

McGrew, D., & Viega, J. (2004). The security and performance of the Galois/Counter Mode (GCM) of operation. In International Conference on Cryptology in India (pp. 343-355). Springer.

McKenney, P. E. (2017). “Is Parallel Programming Hard, And, If So, What Can You Do About It?”. Section 6.1: Queuing.

McKenney, P. E. (2017). “Is Parallel Programming Hard, And, If So, What Can You Do About It?”. Section 6.1: Queuing.

MDN Web Docs. (2023). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Medium Algorithms Topic. (2023). https://medium.com/tag/algorithms

Mehlhorn, K., & Sanders, P. (2008). Algorithms and Data Structures: The Basic Toolbox. Springer.

Meinshausen, N. (2006). Quantile regression forests. Journal of Machine Learning Research, 7(Jun), 983-999.

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of Applied Cryptography. CRC Press. Chapter 2.6: Chinese Remainder Theorem.

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of Applied Cryptography. CRC Press. Chapter 4.4: Prime Number Generation.

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of Applied Cryptography. CRC Press. Chapter 8: Public-Key Encryption.

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of Applied Cryptography. CRC Press.

Menezes, A. J., Van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of Applied Cryptography. CRC Press.

Metwally, A., Agrawal, D., & El Abbadi, A. (2005). “Efficient computation of frequent and top-k elements in data streams”. International Conference on Database Theory, 398-412.

Michael, M. M. (2004). “Hazard pointers: Safe memory reclamation for lock-free objects”. IEEE Transactions on Parallel and Distributed Systems, 15(6), 491-504.

Michael, M. M., & Scott, M. L. (1996). “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms”. Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC ’96), pp. 267-275.

Michael, M. M., & Scott, M. L. (1996). “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms”. Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC ’96), pp. 267-275.

Microsoft Academic. (2023). https://academic.microsoft.com/

Miller, G. L. (1976). Riemann’s Hypothesis and Tests for Primality. Journal of Computer and System Sciences, 13(3), 300-317.

Miner, D., & Shook, A. (2012). “MapReduce Design Patterns”. O’Reilly Media.

Minsky, M., & Papert, S. (1969). Perceptrons: An Introduction to Computational Geometry. MIT Press.

MIT 6.006 Introduction to Algorithms. (2023). https://www.youtube.com/playlist?list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb

MIT OpenCourseWare. (2023). 6.006 Introduction to Algorithms. https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/

MIT OpenCourseWare. (2023). 6.006 Introduction to Algorithms. https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-fall-2011/

Mitchell, T. M. (1997). Machine Learning. McGraw-Hill.

Mitzenmacher, M., & Upfal, E. (2017). Probability and Computing: Randomized Algorithms and Probabilistic Analysis (2nd ed.). Cambridge University Press.

Möller, B. (2003). Improved techniques for fast exponentiation. In Information Security and Cryptology–ICISC 2002 (pp. 298-312). Springer.

Montgomery, D. C., Peck, E. A., & Vining, G. G. (2012). “Introduction to Linear Regression Analysis” (5th ed.). Wiley.

Montgomery, P. L. (1985). Modular multiplication without trial division. Mathematics of Computation, 44(170), 519-521.

Morin, P. (2013). Open Data Structures: An Introduction. AU Press.

Mount, D. (2023). Computational Geometry Resources. https://www.cs.umd.edu/~mount/

Mount, D. M. (2010). ANN Programming Manual. Department of Computer Science, University of Maryland.

Mullner, D. (2011). “Modern hierarchical, agglomerative clustering algorithms”. arXiv preprint arXiv:1109.2378.

Murphy, K. P. (2012). “Machine Learning: A Probabilistic Perspective”. MIT Press.

Murtagh, F., & Contreras, P. (2012). “Algorithms for hierarchical clustering: an overview”. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2(1), 86-97.

Muthukrishnan, S. (2005). “Data streams: Algorithms and applications”. Foundations and Trends in Theoretical Computer Science, 1(2), 117-236.

mycodeschool. (2023). https://www.youtube.com/user/mycodeschool

NeetCode. (2023). Coding Interview Preparation. https://neetcode.io/

NetworkX Documentation. (2023). Network Analysis in Python. https://networkx.org/documentation/stable/

Ng, A. Y., & Jordan, M. I. (2002). “On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes”. Advances in Neural Information Processing Systems, 14, 841-848.

Nielsen, M. A., & Chuang, I. L. (2010). Quantum Computation and Quantum Information. Cambridge University Press.

NIST Special Publication 800-38D. (2007). Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. National Institute of Standards and Technology.

NIST. (2013). Digital Signature Standard (DSS). FIPS PUB 186-4. National Institute of Standards and Technology.

Novikoff, A. B. (1962). On convergence proofs on perceptrons. Proceedings of the Symposium on the Mathematical Theory of Automata, 12, 615-622.

Nussbaumer, H. J. (1982). Fast Fourier transform and convolution algorithms. Springer-Verlag.

Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., & Lomet, D. (1994). “AlphaSort: A RISC machine sort”. ACM SIGMOD Record, 23(2), 233-242.

O’Hearn, Z. W. (2014). BLAKE2: “Harder, Better, Faster, Stronger” Than MD5. Black Hat USA.

O’Neil, P., Cheng, E., Gawlick, D., & O’Neil, E. (1996). “The log-structured merge-tree (LSM-tree)”. Acta Informatica, 33(4), 351-385.

O’Rourke, J. (1998). “Computational Geometry in C” (2nd ed.). Cambridge University Press. - Практичний підручник з реалізаціями алгоритмів.

O’Rourke, J. (1998). “Computational Geometry in C” (2nd ed.). Cambridge University Press.

OEIS. (2023). The On-Line Encyclopedia of Integer Sequences. https://oeis.org/

OI Wiki. (2023). Olympiad in Informatics Wiki. https://oi-wiki.org/en/

Okasaki, C. (1998). Purely Functional Data Structures. Cambridge University Press.

Oliveira e Silva, T., Herzog, S., & Pardi, S. (2014). Empirical verification of the even Goldbach conjecture and computation of prime gaps up to 4×10^18. Mathematics of Computation, 83(288), 2033-2060.

Ongaro, D. (2014). “Consensus: Bridging Theory and Practice”. PhD thesis, Stanford University.

Ongaro, D., & Ousterhout, J. (2014). “In Search of an Understandable Consensus Algorithm”. Proceedings of the 2014 USENIX Annual Technical Conference, 305-320.

OpenDSA. (2023). Interactive Data Structures and Algorithms. https://opendsa-server.cs.vt.edu/

Oppenheim, A. V., & Schafer, R. W. (2009). Discrete-Time Signal Processing (3rd ed.). Prentice Hall.

Ouyang, L., et al. (2022). Training language models to follow instructions with human feedback. NeurIPS.

Paar, C., & Pelzl, J. (2010). Understanding Cryptography: A Textbook for Students and Practitioners. Springer.

Papadias, D., Theodoridis, Y., Sellis, T., & Egenhofer, M. J. (1995). “Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees”. ACM SIGMOD Record, 24(2), 92-103.

Papadimitriou, C. (2023). Columbia CS Page. http://www.cs.columbia.edu/~christos/

Papadimitriou, C.H. “Computational Complexity”. Addison-Wesley, 1994.

Papers We Love. (2023). Computer Science Papers Repository. https://paperswelove.org/

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. ICML.

PathFinding.js. (2023). https://qiao.github.io/PathFinding.js/visual/

Pearson, K. (1901). On Lines and Planes of Closest Fit to Systems of Points in Space. Philosophical Magazine, 2(11), 559-572.

Pedregosa, F., et al. (2011). “Scikit-learn: Machine learning in Python”. Journal of Machine Learning Research, 12, 2825-2830.

Pelleg, D., & Moore, A. W. (2000). “X-means: Extending k-means with efficient estimation of the number of clusters”. Proceedings of the Seventeenth International Conference on Machine Learning, 727-734.

Peters, H., Schulz-Hildebrandt, O., & Luttenberger, N. (2012). “Fast in-place sorting with CUDA based on bitonic sort”. Parallel Processing and Applied Mathematics, 403-410.

Pezzotti, N., et al. (2017). Approximated and user steerable tSNE for progressive visual analytics. IEEE Transactions on Visualization and Computer Graphics, 23(7), 1739-1752.

Pharr, M., Jakob, W., & Humphreys, G. (2016). Physically Based Rendering: From Theory to Implementation (3rd ed.). Morgan Kaufmann.

PHP Data Structures. (2023). https://github.com/php-ds/ext-ds

PHP Manual. (2023). https://www.php.net/manual/en/

Platt, J. (1998). Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines. Microsoft Research Technical Report.

Pluralsight. (2023). Algorithms and Data Structures Path. https://www.pluralsight.com/paths/algorithms-and-data-structures

Pramp. (2023). Data Structures and Algorithms Practice. https://www.pramp.com/

Preparata, F. P., & Hong, S. J. (1977). “Convex hulls of finite sets of points in two and three dimensions”. Communications of the ACM, 20(2), 87-93.

Preparata, F. P., & Shamos, M. I. (1985). “Computational Geometry: An Introduction”. Springer-Verlag. - Один з перших систематичних підручників з обчислювальної геометрії.

Preparata, F. P., & Shamos, M. I. (1985). “Computational Geometry: An Introduction”. Springer-Verlag.

Prim, R. C. (1957). Shortest connection networks and some generalizations. Bell System Technical Journal, 36(6), 1389-1401.

Princeton University. (2023). Algorithms, Part I and II. https://www.coursera.org/learn/algorithms-part1

Pritchard, P. (1987). Linear prime-number sieves: A family tree. Science of Computer Programming, 9(1), 17-35.

Pritchard, P. (1994). Improved incremental prime number sieves. In Algorithmic Number Theory Symposium (pp. 280-288).

Programming Throwdown. (2023). https://www.programmingthrowdown.com/

Project Euler. (2023). Mathematical/Computer Programming Problems. https://projecteuler.net/

Prokhorenkova, L., Gusev, G., Vorobev, A., Dorogush, A. V., & Gulin, A. (2018). CatBoost: unbiased boosting with categorical features. Advances in neural information processing systems, 31.

Queue Visualizer. (2023). https://queue-visualizer.netlify.app/

Quinlan, J. R. (1986). Induction of decision trees. Machine learning, 1(1), 81-106.

Quinlan, J. R. (1993). C4.5: Programs for machine learning. Morgan Kaufmann Publishers.

Quinn, M. J. (2003). Parallel Programming in C with MPI and OpenMP. McGraw-Hill.

Quora Algorithms Topic. (2023). https://www.quora.com/topic/Algorithms

Rabin, M. O. (1980). Probabilistic algorithm for testing primality. Journal of Number Theory, 12(1), 128-138.

Rader, C. (1968). Discrete Fourier transforms when the number of data samples is prime. Proceedings of the IEEE, 56(6), 1107-1108.

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training.

Raju, P., Kadekodi, R., Chidambaram, V., & Abraham, I. (2017). “PebblesDB: Building key-value stores using fragmented log-structured merge trees”. Proceedings of the 26th Symposium on Operating Systems Principles, 497-514.

Red Hat. (2021). “etcd: A distributed, reliable key-value store”. Official etcd documentation.

Red-Black Tree Visualization. (2023). https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Red-Black Tree Visualization. (2023). https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

Reddit r/algorithms. (2023). https://www.reddit.com/r/algorithms/

Reddit r/datastructures. (2023). https://www.reddit.com/r/datastructures/

Reddit r/learnprogramming. (2023). https://www.reddit.com/r/learnprogramming/

Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of naive bayes text classifiers. In Proceedings of the 20th international conference on machine learning.

ResearchGate. (2023). https://www.researchgate.net/

Riesel, H. (1994). Prime Numbers and Computer Methods for Factorization (2nd ed.). Birkhäuser. Chapter 2: The Sieve of Eratosthenes.

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126.

Rogers, H. “Theory of Recursive Functions and Effective Computability”. MIT Press, 1987.

Rokach, L., & Maimon, O. (2014). Data mining with decision trees: theory and applications. World Scientific.

Rosen, K. H. (2018). Elementary Number Theory and Its Applications (6th ed.). Pearson.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386-408.

Rosetta Code. (2023). https://rosettacode.org/wiki/Rosetta_Code

Roughgarden, T. (2017). Algorithms Illuminated: Part 1-4. Soundlikeyourself Publishing.

Roughgarden, T. (2023). Theory Blog. https://timroughgarden.org/

Roy, P., Khan, A., & Alonso, G. (2016). “Augmented sketch: Faster and more accurate stream processing”. Proceedings of the 2016 International Conference on Management of Data, 1449-1463.

Ruby Algorithms. (2023). https://github.com/sagivo/algorithms

Ruby Documentation. (2023). https://ruby-doc.org/core-3.0.0/

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

Rust Documentation. (2023). https://doc.rust-lang.org/book/

Rust std::collections. (2023). https://doc.rust-lang.org/std/collections/

Saarinen, M. J. O. (2015). BLAKE2 tree mode for wide-pipe hash functions. IACR Cryptology ePrint Archive.

SafeCurves: choosing safe curves for elliptic-curve cryptography. (2023). Retrieved from https://safecurves.cr.yp.to/

Salzberg, B. (1990). “Merging sorted runs using large main memory”. Acta Informatica, 27(3), 195-215.

Samet, H. (2006). Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers.

Samet, H. (2006). Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann.

Sammer, E. (2012). “Hadoop Operations”. O’Reilly Media.

Sanders, P., Schlag, S., & Müller, I. (2013). “Communication efficient algorithms for fundamental big data problems”. IEEE International Conference on Big Data.

Satish, N., Harris, M., & Garland, M. (2009). “Designing efficient sorting algorithms for manycore GPUs”. IEEE International Symposium on Parallel & Distributed Processing.

Satish, N., Harris, M., & Garland, M. (2009). “Designing efficient sorting algorithms for manycore GPUs”. IEEE International Symposium on Parallel & Distributed Processing.

Scala Collections. (2023). https://docs.scala-lang.org/overviews/collections-2.13/introduction.html

Schmidt, J. (2017). Performance analysis of cryptographic hash functions. Technical Report, University of Technology.

Schölkopf, B., & Smola, A. J. (2002). Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press.

Schubert, E., Sander, J., Ester, M., Kriegel, H. P., & Xu, X. (2017). “DBSCAN revisited, revisited: why and how you should (still) use DBSCAN”. ACM Transactions on Database Systems, 42(3), 1-21.

Sculley, D. (2010). “Web-scale k-means clustering”. Proceedings of the 19th International Conference on World Wide Web, 1177-1178.

Seber, G. A., & Lee, A. J. (2003). “Linear Regression Analysis” (2nd ed.). Wiley-Interscience.

SEC 2: Recommended Elliptic Curve Domain Parameters. (2010). Standards for Efficient Cryptography Group.

Sedgewick, R. (2023). Personal Website. https://www.cs.princeton.edu/~rs/

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

Seidel, R., & Aragon, C. R. (1996). Randomized search trees. Algorithmica, 16(4-5), 464-497.

Sellis, T., Roussopoulos, N., & Faloutsos, C. (1987). “The R+-Tree: A Dynamic Index for Multi-Dimensional Objects”. Proceedings of the 13th International Conference on Very Large Data Bases, 507-518.

Semantic Scholar. (2023). https://www.semanticscholar.org/

Shirley, P., & Marschner, S. (2009). Fundamentals of Computer Graphics (3rd ed.). A K Peters/CRC Press.

Shlens, J. (2014). A Tutorial on Principal Component Analysis. arXiv preprint arXiv:1404.1100.

Shoup, V. (2009). A Computational Introduction to Number Theory and Algebra (2nd ed.). Cambridge University Press. Chapter 10: Primality Testing.

Shoup, V. (2009). A Computational Introduction to Number Theory and Algebra (2nd ed.). Cambridge University Press. Chapter 2.8: Chinese Remainder Theorem.

Shoup, V. (2009). A Computational Introduction to Number Theory and Algebra (2nd ed.). Cambridge University Press. Section 5.4: The Sieve of Eratosthenes.

Shoup, V. (2009). A Computational Introduction to Number Theory and Algebra (2nd ed.). Cambridge University Press.

Sibson, R. (1973). “SLINK: an optimally efficient algorithm for the single-link cluster method”. The Computer Journal, 16(1), 30-34.

Silverman, J. H. (2009). The Arithmetic of Elliptic Curves (2nd ed.). Springer.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

Sipser, M. (2023). Theory of Computation. http://math.mit.edu/~sipser/

Sipser, M. “Introduction to the Theory of Computation” (3rd edition). Cengage Learning, 2012.

Sit, E., & Morris, R. (2002). “Security considerations for peer-to-peer distributed hash tables”. International Workshop on Peer-to-Peer Systems, 261-269.

Skiena, S. S. (2020). The Algorithm Design Manual (3rd ed.). Springer.

Slack Algorithm Communities. (2023). https://slofile.com/category/Programming

Sleator, D. D., & Tarjan, R. E. (1985). Self-adjusting binary search trees. Journal of the ACM, 32(3), 652-686.

Smart, N. (2016). Cryptography Made Simple. Springer. Chapter 15: The RSA Algorithm.

Software Engineering Daily. (2023). https://softwareengineeringdaily.com/

Sokal, R. R., & Rohlf, F. J. (1962). “The comparison of dendrograms by objective methods”. Taxon, 11(2), 33-40.

SoloLearn. (2023). Data Structures Course. https://www.sololearn.com/

Sorenson, J. (1998). Trading time for space in prime number sieves. In Algorithmic Number Theory (pp. 179-194). Springer.

Sorting Visualizer. (2023). https://sorting-visualizer.netlify.app/

SPOJ. (2023). Sphere Online Judge. https://www.spoj.com/

SpringerLink. (2023). https://link.springer.com/

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. JMLR.

Stack Exchange Computer Science. (2023). https://cs.stackexchange.com/

Stack Overflow. (2023). Data Structures and Algorithms Q&A. https://stackoverflow.com/questions/tagged/data-structures

Stack Overflow. (2023). Data Structures and Algorithms Q&A. https://stackoverflow.com/questions/tagged/data-structures

Stallings, W. (2017). Cryptography and Network Security: Principles and Practice (7th ed.). Pearson. Chapter 8: More Number Theory.

Stanford University. (2023). CS161: Design and Analysis of Algorithms. https://web.stanford.edu/class/cs161/

Steiner, M., En-Najjary, T., & Biersack, E. W. (2009). “Long term study of peer behavior in the KAD DHT”. IEEE/ACM Transactions on Networking, 17(6), 1737-1750.

Steinwart, I., & Christmann, A. (2008). Support Vector Machines. Springer Science & Business Media.

Stinson, D. R. (2005). Cryptography: Theory and Practice (3rd ed.). Chapman and Hall/CRC.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). “Chord: A scalable peer-to-peer lookup service for internet applications”. ACM SIGCOMM Computer Communication Review, 31(4), 149-160.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M. F., Dabek, F., & Balakrishnan, H. (2003). “Chord: A scalable peer-to-peer lookup protocol for internet applications”. IEEE/ACM Transactions on Networking, 11(1), 17-32.

Strassen, V. (1969). Gaussian elimination is not optimal. Numerische Mathematik, 13(4), 354-356.

Strobl, C., Boulesteix, A. L., Zeileis, A., & Hothorn, T. (2007). Bias in random forest variable importance measures. BMC bioinformatics, 8(1), 1-10.

Structy. (2023). Data Structures Course. https://structy.net/

Stutzbach, D., & Rejaie, R. (2006). “Understanding churn in peer-to-peer networks”. Proceedings of the 6th ACM SIGCOMM conference on Internet measurement, 189-202.

Suffern, K. (2007). Ray Tracing from the Ground Up. A K Peters.

Sundell, H., & Tsigas, P. (2005). “Fast and lock-free concurrent priority queues for multi-thread systems”. Journal of Parallel and Distributed Computing, 65(5), 609-627.

Sundell, H., & Tsigas, P. (2005). “Fast and lock-free concurrent priority queues for multi-thread systems”. Journal of Parallel and Distributed Computing, 65(5), 609-627.

SUPERCOP. (2023). System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives. BLAKE2 benchmarks.

Swift Algorithm Club. (2023). https://github.com/raywenderlich/swift-algorithm-club

Swift Documentation. (2023). https://swift.org/documentation/

Szegedy, C., Liu, W., Jia, Y., et al. (2015). Going deeper with convolutions. CVPR.

Tarjan, R. E. (1983). Data Structures and Network Algorithms. SIAM.

Tarjan, R. E. (1985). Amortized computational complexity. SIAM Journal on Algebraic Discrete Methods, 6(2), 306-318.

Tarjan, R. E. (2023). Personal Website. https://www.cs.princeton.edu/~ret/

Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2020). Efficient transformers: A survey. arXiv preprint arXiv:2009.06732.

Tech With Tim. (2023). https://www.youtube.com/channel/UC4JX40jDee_tINbkjycV4Sg

Techie Delight. (2023). 500+ Data Structures and Algorithms Problems. https://www.techiedelight.com/

The Algorithms. (2023). Open Source Algorithm Implementations. https://github.com/TheAlgorithms

The Algorithms. (2023). Open Source Algorithm Implementations. https://github.com/TheAlgorithms

The Changelog. (2023). https://changelog.com/podcast

The Morning Paper. (2023). Daily CS Paper Reviews. https://blog.acolyer.org/

Tibshirani, R. (1996). “Regression shrinkage and selection via the lasso”. Journal of the Royal Statistical Society, Series B, 58(1), 267-288.

TiKV Authors. (2019). “Deep Dive TiKV: Distributed Transactions”. PingCAP blog series on Raft in production.

Ting, D. (2018). “Count-min: Optimal estimation and tight error bounds using empirical error distributions”. Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 2319-2328.

Tipping, M. E., & Bishop, C. M. (1999). Probabilistic principal component analysis. Journal of the Royal Statistical Society: Series B, 61(3), 611-622.

TopCoder. (2023). Algorithm Tutorials. https://www.topcoder.com/thrive/tracks?track=Competitive%20Programming

TopCoder. (2023). Algorithm Tutorials. https://www.topcoder.com/thrive/tracks?track=Competitive%20Programming

Touvron, H., et al. (2023). LLaMA: Open and efficient foundation language models. arXiv preprint.

Tree Visualizer. (2023). https://tree-visualizer.netlify.app/

Treiber, R. K. (1986). “Systems programming: Coping with parallelism”. Technical Report RJ 5118, IBM Almaden Research Center.

Turing, A.M. “On Computable Numbers, with an Application to the Entscheidungsproblem”. Proceedings of the London Mathematical Society, 1936.

Tushar Roy - Coding Made Simple. (2023). https://www.youtube.com/user/tusharroy2525

Udacity. (2023). Data Structures and Algorithms Nanodegree. https://www.udacity.com/course/data-structures-and-algorithms-nanodegree–nd256

Udacity. (2023). Data Structures and Algorithms Nanodegree. https://www.udacity.com/course/data-structures-and-algorithms-nanodegree–nd256

Udemy. (2023). Master the Coding Interview: Data Structures + Algorithms. https://www.udemy.com/course/master-the-coding-interview-data-structures-algorithms/

USACO Guide. (2023). Competitive Programming Resources. https://usaco.guide/

UVa Online Judge. (2023). https://onlinejudge.org/

van der Maaten, L. (2009). Learning a parametric embedding by preserving local structure. In Artificial Intelligence and Statistics (pp. 384-391).

van der Maaten, L. (2014). Accelerating t-SNE using tree-based algorithms. Journal of Machine Learning Research, 15(93), 3221-3245.

van der Maaten, L., & Hinton, G. (2008). Visualizing data using t-SNE. Journal of Machine Learning Research, 9(86), 2579-2605.

Van Loan, C. (1992). Computational Frameworks for the Fast Fourier Transform. SIAM.

Van Renesse, R., & Altinbuken, D. (2015). "Paxos Made Moderately Complex". ACM Computing Surveys, 47(3), 1-36.

Vapnik, V. (1995). The Nature of Statistical Learning Theory. Springer-Verlag.

Vaswani, A., et al. (2017). Attention is all you need. NeurIPS.

Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). Attention is all you need. NeurIPS.

VisuAlgo. (2023). Graph Algorithms Visualization. https://visualgo.net/en/graphds

Visualgo. (2023). Visualising Data Structures and Algorithms Through Animation. https://visualgo.net/

Vitter, J. S. (2008). Algorithms and Data Structures for External Memory. Now Publishers.

Vuillemin, J. (1978). A data structure for manipulating priority queues. Communications of the ACM, 21(4), 309-315.

Walter, C. D. (2002). Longer keys may facilitate side channel attacks. In Selected Areas in Cryptography (pp. 42-57). Springer.

Wann Jensen, H. (2001). Realistic Image Synthesis Using Photon Mapping. A K Peters.

Ward Jr, J. H. (1963). "Hierarchical grouping to optimize an objective function". Journal of the American Statistical Association, 58(301), 236-244.

Washington, L. C. (2008). Elliptic Curves: Number Theory and Cryptography (2nd ed.). Chapman & Hall/CRC.

Wattenberg, M., Viégas, F., & Johnson, I. (2016). How to use t-SNE effectively. Distill, 1(10), e2.

Wayne, K. (2023). Princeton CS Page. https://www.cs.princeton.edu/~wayne/

Wei, J., et al. (2022). Emergent abilities of large language models. TMLR.

Weiss, M. A. (2013). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.

White, T. (2015). “Hadoop: The Definitive Guide” (4th ed.). O’Reilly Media.

WikiBooks Algorithms. (2023). https://en.wikibooks.org/wiki/Algorithms

WikiBooks Data Structures. (2023). https://en.wikibooks.org/wiki/Data_Structures

William Fiset. (2023). https://www.youtube.com/user/purpongie

Williams, A. (2019). “C++ Concurrency in Action” (2nd Edition). Manning Publications. Chapter 7: Designing lock-free concurrent data structures.

Williams, A. (2019). “C++ Concurrency in Action” (2nd Edition). Manning Publications. Chapter 7: Designing lock-free concurrent data structures.

Williams, J. W. J. (1964). Algorithm 232: Heapsort. Communications of the ACM, 7(6), 347-348.

Williams, V. V. (2012). Multiplying matrices faster than Coppersmith-Winograd. In Proceedings of STOC 2012 (pp. 887-898).

Winograd, S. (1971). On multiplication of 2×2 matrices. Linear Algebra and its Applications, 4(4), 381-388.

Winograd, S. (1978). On computing the discrete Fourier transform. Mathematics of computation, 32(141), 175-199.

Wolfram MathWorld. (2023). https://mathworld.wolfram.com/

Wood, G. (2014). “Ethereum: A secure decentralised generalised transaction ledger”. Ethereum project yellow paper, 151(2014), 1-32.

Wu, X., Xu, Y., Shao, Z., & Jiang, S. (2015). “LSM-trie: An LSM-tree-based ultra-large key-value store for small data items”. 2015 USENIX Annual Technical Conference, 71-82.

Zeiler, M. D., & Fergus, R. (2014). Visualizing and understanding convolutional networks. ECCV.

Zhang, H. (2004). The optimality of naive Bayes. In Proceedings of the 17th international Florida artificial intelligence research society conference.

Zhang, Y., & Zhang, Y. (2019). “Count-Min Sketch and its applications: A survey”. IEEE Access, 7, 174982-175000.

Zheng, F., & Zhu, K. (2015). “Efficient external sorting on flash memory embedded devices”. International Journal of Database Management Systems, 7(1), 21-38.

Zou, H., & Hastie, T. (2005). “Regularization and variable selection via the elastic net”. Journal of the Royal Statistical Society, Series B, 67(2), 301-320.

Додаток 1

Як не виконувати лабораторні роботи

Лабораторна робота № 3

1. Назва роботи: Реалізація та аналіз алгоритму бінарного пошуку

2. Мета роботи Набути практичних навичок у реалізації алгоритму бінарного пошуку. Емпірично дослідити його поведінку в найкращому, середньому та найгіршому випадках. Порівняти ефективність з лінійним пошуком.

3. Теоретичні відомості

3.1. Опис алгоритму Бінарний пошук — це ефективний алгоритм пошуку елемента у відсортованому масиві. Алгоритм працює за принципом “розділяй і володарюй”, на кожному кроці зменшуючи область пошуку вдвічі.

3.2. Принцип роботи

Порівнюємо шуканий елемент із середнім елементом масиву Якщо елементи рівні — пошук завершено Якщо шуканий елемент менший — продовжуємо пошук у лівій половині Якщо шуканий елемент більший — продовжуємо пошук у правій половині Повторюємо процес до знаходження елемента або вичерпання області пошуку

3.3. Складність алгоритму

Часова складність: O(log n) для всіх випадків Просторова складність: O(1) для ітеративної версії

4. Хід виконання роботи

4.1. Реалізація алгоритму

```
import time import random
def binary_search(arr, key): """ Функція бінарного пошуку елемента в
відсортованому масиві.
Параметри:
arr (list): Відсортований масив для пошуку
key: Елемент, який шукаємо

Повертає:
int: Індекс знайденого елемента або -1, якщо елемент не знайдено
```

```
"""
left = 0
right = len(arr) - 1
comparisons = 0  # Лічильник порівнянь для аналізу

while left <= right:
    comparisons += 1
    mid = (left + right) // 2

    if arr[mid] == key:
        print(f"  Елемент знайдено після {comparisons} порівнянь")
        return mid
    elif arr[mid] < key:
        left = mid + 1
    else:
        right = mid - 1

print(f"  Елемент не знайдено після {comparisons} порівнянь")
return -1
def binary_search_recursive(arr, key, left=0, right=None): “”” Рекурс
ивна реалізація бінарного пошуку.
Параметри:
arr (list): Відсортований масив для пошуку
key: Елемент, який шукаємо
left (int): Лівий індекс області пошуку
right (int): Правий індекс області пошуку

Повертає:
int: Індекс знайденого елемента або -1, якщо елемент не знайдено
"""
if right is None:
    right = len(arr) - 1

if left > right:
    return -1

mid = (left + right) // 2

if arr[mid] == key:
    return mid
elif arr[mid] < key:
    return binary_search_recursive(arr, key, mid + 1, right)
else:
    return binary_search_recursive(arr, key, left, mid - 1)
```

```
def linear_search(arr, key): “”” Функція лінійного пошуку для порівня
ння ефективності.
Параметри:
arr (list): Масив для пошуку
key: Елемент, який шукаємо

Повертає:
int: Індекс знайденого елемента або -1, якщо елемент не знайдено
"""
comparisons = 0
for i in range(len(arr)):
    comparisons += 1
    if arr[i] == key:
        print(f"  Лінійний пошук: знайдено після {comparisons} порівн
янь")
        return i
print(f"  Лінійний пошук: не знайдено після {comparisons} порівнянь")
return -1
def measure_time(func, arr, key): “”” Вимірює час виконання функції п
ошуку.
Параметри:
func: Функція пошуку
arr (list): Масив для пошуку
key: Елемент, який шукаємо

Повертає:
tuple: (результат, час_виконання)
"""
start_time = time.perf_counter()
result = func(arr, key)
end_time = time.perf_counter()
return result, (end_time - start_time) * 1000  # час в мілісекундах
def main(): “””Головна функція програми.”””
print("=" * 60)
print("ЛАБОРАТОРНА РОБОТА: БІНАРНИЙ ПОШУК")
print("=" * 60)

# Створення тестового масиву
test_array = [3, 5, 8, 12, 17, 23, 29, 34, 42, 51, 67, 73, 88, 95, 99
]
print(f"\nТестовий відсортований масив:")
print(f"Масив: {test_array}")
print(f"Розмір масиву: {len(test_array)}")
print(f"Індекси: {list(range(len(test_array)))}")
```

```
print("\n" + "-" * 60)
print("ТЕСТУВАННЯ РІЗНИХ ВИПАДКІВ")
print("-" * 60)

# 1. Найкращий випадок - елемент в центрі
print("\n1. НАЙКРАЩИЙ ВИПАДОК (елемент в центрі масиву):")
middle_element = test_array[len(test_array) // 2]
print(f"Шукаємо елемент: {middle_element}")
index = binary_search(test_array, middle_element)
if index != -1:
    print(f"  Результат: елемент {middle_element} знайдено на позиції
{index}")

# 2. Найгірший випадок - елемент на початку
print("\n2. НАЙГІРШИЙ ВИПАДОК #1 (елемент на початку масиву):")
first_element = test_array[0]
print(f"Шукаємо елемент: {first_element}")
index = binary_search(test_array, first_element)
if index != -1:
    print(f"  Результат: елемент {first_element} знайдено на позиції
{index}")

# 3. Найгірший випадок - елемент в кінці
print("\n3. НАЙГІРШИЙ ВИПАДОК #2 (елемент в кінці масиву):")
last_element = test_array[-1]
print(f"Шукаємо елемент: {last_element}")
index = binary_search(test_array, last_element)
if index != -1:
    print(f"  Результат: елемент {last_element} знайдено на позиції {
index}")

# 4. Випадок відсутності елемента
print("\n4. ЕЛЕМЕНТ ВІДСУТНІЙ В МАСИВІ:")
missing_element = 100
print(f"Шукаємо елемент: {missing_element}")
index = binary_search(test_array, missing_element)
if index == -1:
    print(f"  Результат: елемент {missing_element} не знайдено (повер
нуто -1)")

# 5. Середній випадок
print("\n5. СЕРЕДНІЙ ВИПАДОК (елемент в довільній позиції):")
random_element = test_array[5]  # елемент 23
print(f"Шукаємо елемент: {random_element}")
index = binary_search(test_array, random_element)
```

```
if index != -1:
    print(f"  Результат: елемент {random_element} знайдено на позиції
{index}")

print("\n" + "-" * 60)
print("ПОРІВНЯННЯ З ЛІНІЙНИМ ПОШУКОМ")
print("-" * 60)

# Порівняння з лінійним пошуком
search_key = 73
print(f"\nШукаємо елемент {search_key}:")
print("\nБінарний пошук:")
binary_result, binary_time = measure_time(binary_search, test_array,
search_key)
print(f"  Час виконання: {binary_time:.4f} мс")

print("\nЛінійний пошук:")
linear_result, linear_time = measure_time(linear_search, test_array,
search_key)
print(f"  Час виконання: {linear_time:.4f} мс")

print("\n" + "-" * 60)
print("ТЕСТУВАННЯ НА ВЕЛИКОМУ МАСИВІ")
print("-" * 60)

# Створення великого масиву для демонстрації ефективності
large_size = 10000
large_array = sorted([random.randint(1, 100000) for _ in range(large_
size)])
search_element = large_array[large_size // 2]

print(f"\nМасив з {large_size} елементів")
print(f"Шукаємо елемент: {search_element}")

print("\nБінарний пошук:")
start = time.perf_counter()
binary_idx = binary_search(large_array, search_element)
binary_large_time = (time.perf_counter() - start) * 1000
print(f"  Час виконання: {binary_large_time:.4f} мс")

print("\nЛінійний пошук:")
start = time.perf_counter()
linear_idx = linear_search(large_array, search_element)
linear_large_time = (time.perf_counter() - start) * 1000
print(f"  Час виконання: {linear_large_time:.4f} мс")
```

```
if linear_large_time > 0:
    speedup = linear_large_time / binary_large_time
    print(f"\nБінарний пошук швидший в {speedup:.1f} разів!")

print("\n" + "-" * 60)
print("ТЕСТУВАННЯ РЕКУРСИВНОЇ РЕАЛІЗАЦІЇ")
print("-" * 60)

print(f"\nРекурсивний бінарний пошук елемента {test_array[7]}:")
index = binary_search_recursive(test_array, test_array[7])
print(f"  Результат: елемент знайдено на позиції {index}")

print("\n" + "=" * 60)
print("КІНЕЦЬ ТЕСТУВАННЯ")
print("=" * 60)
if name == “main”: main()
```

4.2. Результати виконання програми Після запуску програми отримано наступні результати:

==========

ТЕСТУВАННЯ АЛГОРИТМІВ ПОШУКУ

==========

Тестовий масив: [3, 7, 12, 19, 25, 33, 42, 56, 67, 78, 89, 95, 102, 115, 128] Розмір масиву: 15

==========

БІНАРНИЙ ПОШУК

==========

1. НАЙКРАЩИЙ ВИПАДОК (середній елемент): Шукаємо елемент: 56 Елемент знайдено за 1 порівнянь Результат: індекс 7

2. НАЙГІРШИЙ ВИПАДОК (перший елемент): Шукаємо елемент: 3 Елемент знайдено за 4 порівнянь Результат: індекс 0

3. НАЙГІРШИЙ ВИПАДОК (останній елемент): Шукаємо елемент: 128 Елемент знайдено за 4 порівнянь Результат: індекс 14

4. СЕРЕДНІЙ ВИПАДОК (довільний елемент): Шукаємо елемент: 33 Елемент знайдено за 3 порівнянь Результат: індекс 5

5. ЕЛЕМЕНТ ВІДСУТНІЙ: Шукаємо елемент: 50 Елемент не знайдено. Виконано 4 порівнянь Результат: -1

5. Аналіз результатів

Ефективність: O(log n) проти O(n) у лінійного пошуку

Масштабованість: На великих масивах різниця стає критичною

Передбачуваність: Максимальна кількість операцій завжди $\log_2(n)$

Обмеження бінарного пошуку

Вимога сортування: Масив повинен бути відсортований Додаткові витрати: Сортування вимагає O(n log n) операцій Доступ до елементів: Потрібен прямий доступ за індексом

6. Висновки У ході виконання лабораторної роботи було успішно реалізовано алгоритм бінарного пошуку та досліджено його поведінку в різних сценаріях.

Основні висновки:

Ефективність алгоритму: Бінарний пошук демонструє логарифмічну складність O(log n), що робить його набагато ефективнішим за лінійний пошук на великих масивах даних.

Практичне застосування: На тестовому масиві з 15 елементів бінарний пошук виконує максимум 4 порівняння, тоді як лінійний може виконати до 15.

Масштабованість: При збільшенні розміру масиву до 1000 елементів, бінарний пошук залишається ефективним (максимум 10 порівнянь), тоді як лінійний може виконати до 1000 порівнянь.

Обмеження: Необхідність попереднього сортування масиву є основним обмеженням бінарного пошуку. Якщо дані часто змінюються, витрати на підтримку сортування можуть нівелювати переваги алгоритму.

Рекомендації щодо використання:

Використовувати бінарний пошук для великих відсортованих масивів Для малих масивів (< 10 елементів) різниця несуттєва Для несортованих даних або при частих змінах краще використовувати інші структури даних (хеш-таблиці)

Лабораторна робота продемонструвала важливість вибору правильного алгоритму залежно від характеристик даних та вимог задачі.

8. Список використаних джерел

Кормен Т., Лейзерсон Ч., Рівест Р., Штайн К. Вступ до алгоритмів. 3-тє видання.

Седжвік Р. Алгоритми на Java. Частини 1-4: Основи, структури даних, сортування, пошук.

Документація Python: https://docs.python.org/3/

Роботу виконав: [Ваше ім’я] Група: [Ваша група] Дата: [Поточна дата]

Лабораторна робота № 4

Лабораторна робота: Реалізація алгоритму експоненційного пошуку

Методичні вказівки

1. Назва

Реалізація алгоритму експоненційного пошуку.

2. Мета

Практичне засвоєння двофазної логіки експоненційного пошуку та його застосування для пошуку в масивах.

3. Завдання

Створити програму, яка реалізує функцію експоненційного пошуку. Програма має продемонструвати знаходження діапазону та подальший виклик двійкового пошуку.

4. Опис

Програма повинна містити реалізацію експоненційного пошуку, яка всередині себе викликає раніше реалізовану функцію двійкового пошуку. Це гарний приклад композиції алгоритмів.

5. Порядок виконання

1. Підготуйте інструменти: Візьміть вашу реалізацію ітеративного двійкового пошуку з попередньої лабораторної роботи.

2. Реалізуйте функцію ExponentialSearch:

- Перевірте перший елемент масиву.

- Реалізуйте цикл while для знаходження верхньої межі bound шляхом її подвоєння. Додайте виведення на екран, щоб бачити, як зростає bound.

- Визначте нижню та верхню межі для двійкового пошуку.

- Викличте вашу функцію двійкового пошуку для знайденого діапазону.

3. Створіть тестовий масив: В основній частині програми створіть великий відсортований масив (напр., 100 елементів).

4. Протестуйте: Перевірте роботу, шукаючи елемент на початку масиву (напр., індекс 5), в середині (індекс 50) та в кінці (індекс 95). Спостерігайте за виведенням, щоб побачити, які діапазони знаходить перший крок алгоритму.

6. Результат

Додаток, що коректно знаходить елементи у відсортованому масиві за допомогою експоненційного пошуку.

ЗВІТ З ЛАБОРАТОРНОЇ РОБОТИ

Тема: Реалізація алгоритму експоненційного пошуку

Виконав: студент групи _____

Перевірив: _____

Дата: _____

1. Теоретичні відомості

Експоненційний пошук (Exponential Search) - це алгоритм пошуку елемента у відсортованому масиві, який поєднує переваги лінійного та двійкового пошуку.

Основні характеристики: - Часова складність: O(log n) - Просторова складність: O(1) - Працює тільки з відсортованими масивами

Алгоритм працює у два етапи: 1. **Фаза експоненційного росту**: Починаючи з індексу 1, подвоюємо індекс (1, 2, 4, 8, 16...) поки не знайдемо елемент більший за шуканий або не досягнемо кінця масиву. 2. **Фаза двійкового пошуку**: Застосовуємо двійковий пошук у знайденому діапазоні.

2. Вихідний код програми

```
def binary_search(arr, left, right, target):
    """
    Ітеративний двійковий пошук у відсортованому масиві

    Args:
        arr: відсортований масив
```

```
        left: ліва межа пошуку
        right: права межа пошуку
        target: шуканий елемент

    Returns:
        індекс елемента або -1, якщо елемент не знайдено
    """
    print(f"  Двійковий пошук в діапазоні [{left}, {right}]")

    while left <= right:
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1


def exponential_search(arr, target):
    """
    Експоненційний пошук елемента у відсортованому масиві

    Args:
        arr: відсортований масив
        target: шуканий елемент

    Returns:
        індекс елемента або -1, якщо елемент не знайдено
    """
    n = len(arr)
    print(f"\nПошук елемента {target} в масиві довжиною {n}")

    # Перевірка першого елемента
    if arr[0] == target:
        print("Елемент знайдено на позиції 0")
        return 0

    # Фаза експоненційного росту
    bound = 1
    print("Фаза експоненційного росту:")
```

```
    while bound < n and arr[bound] <= target:
        print(f"  bound = {bound}, arr[{bound}] = {arr[bound]}")
        bound *= 2

    print(f"  Фінальне bound = {bound}")

    # Визначення меж для двійкового пошуку
    left = bound // 2
    right = min(bound - 1, n - 1)

    print(f"Визначені межі для двійкового пошуку: [{left}, {right}]")

    # Фаза двійкового пошуку
    result = binary_search(arr, left, right, target)

    if result != -1:
        print(f"Елемент {target} знайдено на позиції {result}")
    else:
        print(f"Елемент {target} не знайдено")

    return result


def create_test_array(size):
    """Створює відсортований масив для тестування"""
    return [i * 2 for i in range(size)]


def main():
    """Основна функція для тестування алгоритму"""
    print("="*60)
    print("ТЕСТУВАННЯ АЛГОРИТМУ ЕКСПОНЕНЦІЙНОГО ПОШУКУ")
    print("="*60)

    # Створення тестового масиву
    test_array = create_test_array(100)
    print(f"Створено тестовий масив: [0, 2, 4, 6, ..., {test_array[-1
]}]")
    print(f"Довжина масиву: {len(test_array)}")

    # Тест 1: Пошук елемента на початку масиву
    print("\n" + "-"*60)
    print("ТЕСТ 1: Пошук елемента на початку масиву")
    target1 = test_array[5]  # елемент з індексом 5
    result1 = exponential_search(test_array, target1)
```

```
    # Тест 2: Пошук елемента в середині масиву
    print("\n" + "-"*60)
    print("ТЕСТ 2: Пошук елемента в середині масиву")
    target2 = test_array[50]  # елемент з індексом 50
    result2 = exponential_search(test_array, target2)

    # Тест 3: Пошук елемента в кінці масиву
    print("\n" + "-"*60)
    print("ТЕСТ 3: Пошук елемента в кінці масиву")
    target3 = test_array[95]  # елемент з індексом 95
    result3 = exponential_search(test_array, target3)

    # Тест 4: Пошук неіснуючого елемента
    print("\n" + "-"*60)
    print("ТЕСТ 4: Пошук неіснуючого елемента")
    target4 = 199  # неіснуючий елемент
    result4 = exponential_search(test_array, target4)

    # Підсумки
    print("\n" + "="*60)
    print("ПІДСУМКИ ТЕСТУВАННЯ:")
    print(f"Тест 1 (початок): шукали {target1}, знайшли на позиції {r
esult1}")
    print(f"Тест 2 (середина): шукали {target2}, знайшли на позиції {
result2}")
    print(f"Тест 3 (кінець): шукали {target3}, знайшли на позиції {re
sult3}")
    print(f"Тест 4 (неіснуючий): шукали {target4}, результат: {result
4}")
    print("="*60)


if __name__ == "__main__":
    main()
```

3. Результати виконання програми

```
============================================================
ТЕСТУВАННЯ АЛГОРИТМУ ЕКСПОНЕНЦІЙНОГО ПОШУКУ
============================================================
Створено тестовий масив: [0, 2, 4, 6, ..., 198]
Довжина масиву: 100

------------------------------------------------------------
ТЕСТ 1: Пошук елемента на початку масиву
```

```
Пошук елемента 10 в масиві довжиною 100
Фаза експоненційного росту:
  bound = 1, arr[1] = 2
  bound = 2, arr[2] = 4
  bound = 4, arr[4] = 8
  bound = 8, arr[8] = 16
  Фінальне bound = 16
Визначені межі для двійкового пошуку: [8, 15]
  Двійковий пошук в діапазоні [8, 15]
Елемент 10 знайдено на позиції 5

------------------------------------------------------------
ТЕСТ 2: Пошук елемента в середині масиву

Пошук елемента 100 в масиві довжиною 100
Фаза експоненційного росту:
  bound = 1, arr[1] = 2
  bound = 2, arr[2] = 4
  bound = 4, arr[4] = 8
  bound = 8, arr[8] = 16
  bound = 16, arr[16] = 32
  bound = 32, arr[32] = 64
  bound = 64, arr[64] = 128
  Фінальне bound = 128
Визначені межі для двійкового пошуку: [64, 99]
  Двійковий пошук в діапазоні [64, 99]
Елемент 100 знайдено на позиції 50

------------------------------------------------------------
ТЕСТ 3: Пошук елемента в кінці масиву

Пошук елемента 190 в масиві довжиною 100
Фаза експоненційного росту:
  bound = 1, arr[1] = 2
  bound = 2, arr[2] = 4
  bound = 4, arr[4] = 8
  bound = 8, arr[8] = 16
  bound = 16, arr[16] = 32
  bound = 32, arr[32] = 64
  bound = 64, arr[64] = 128
  Фінальне bound = 128
Визначені межі для двійкового пошуку: [64, 99]
  Двійковий пошук в діапазоні [64, 99]
Елемент 190 знайдено на позиції 95
```

```
------------------------------------------------------------
ТЕСТ 4: Пошук неіснуючого елемента

Пошук елемента 199 в масиві довжиною 100
Фаза експоненційного росту:
  bound = 1, arr[1] = 2
  bound = 2, arr[2] = 4
  bound = 4, arr[4] = 8
  bound = 8, arr[8] = 16
  bound = 16, arr[16] = 32
  bound = 32, arr[32] = 64
  bound = 64, arr[64] = 128
  Фінальне bound = 128
Визначені межі для двійкового пошуку: [64, 99]
  Двійковий пошук в діапазоні [64, 99]
Елемент 199 не знайдено

============================================================
ПІДСУМКИ ТЕСТУВАННЯ:
Тест 1 (початок): шукали 10, знайшли на позиції 5
Тест 2 (середина): шукали 100, знайшли на позиції 50
Тест 3 (кінець): шукали 190, знайшли на позиції 95
Тест 4 (неіснуючий): шукали 199, результат: -1
============================================================
```

4. Аналіз результатів

Спостереження:

1. **Ефективність для елементів на початку**: При пошуку елемента з індексом 5, алгоритм швидко знайшов діапазон [8, 15], що значно менше за весь масив.

2. **Робота з елементами в середині та кінці**: Для елементів з індексами 50 та 95, алгоритм визначив однаковий діапазон [64, 99], що демонструє логарифмічну природу росту bound.

3. **Експоненційний ріст**: Значення bound зростає як 1, 2, 4, 8, 16, 32, 64, 128, що підтверджує експоненційну природу першої фази.

4. **Обробка неіснуючих елементів**: Алгоритм коректно обробляє випадок, коли елемент відсутній у масиві.

Переваги алгоритму:

- **Оптимальність для малих індексів**: Особливо ефективний, коли шуканий елемент знаходиться близько до початку масиву

- **Універсальність**: Працює для масивів будь-якого розміру без попереднього знання їх довжини

- **Композиція алгоритмів**: Демонструє ефективне поєднання двох різних підходів

5. Висновки

У ході виконання лабораторної роботи було успішно реалізовано алгоритм експоненційного пошуку. Програма демонструє двофазну природу алгоритму: спочатку визначається діапазон можливого розташування елемента через експоненційне збільшення індексу, а потім застосовується двійковий пошук у знайденому діапазоні.

Тестування підтвердило коректність роботи алгоритму для різних випадків: елементів на початку, в середині та в кінці масиву, а також для неіснуючих елементів. Алгоритм показав свою ефективність, особливо для пошуку елементів з малими індексами.

Практична реалізація допомогла глибше зрозуміти принципи композиції алгоритмів та переваги гібридних підходів у розв’язанні задач пошуку.