

Міністерство освіти і науки України

**Національний технічний університет України «Київський політехнічний
інститут імені Ігоря Сікорського”**

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Проектування алгоритмів

КУРС ЛЕКЦІЙ

для студентів спеціальності 121

укладач ст. в. Головченко Максим Миколайович

Київ 2023

ЗМІСТ

1 ТЕМА 1 – АЛГОРИТМИ СОРТУВАННЯ.....	7
1.1 Алгоритми зовнішнього сортування	7
1.1.1 Пряме злиття.....	8
1.1.2 Природне злиття	9
1.1.3 Збалансоване багатошляхове злиття	10
1.1.4 Багатофазне сортування	14
1.1.5 Покращення ефективності зовнішнього сортування за рахунок використання оперативної пам'яті	18
2 ТЕМА 2 АЛГОРИТМИ ПОШУКУ.....	19
2.1 Спрощені задачі	19
2.2 Реальні задачі.....	23
2.3 Пошук рішень.....	26
2.4 Вимірювання продуктивності рішення задачі	30
2.5 Стратегії неінформативного пошуку	32
2.5.1 Пошук в ширину.....	32
2.5.2 Пошук в глибину	36
2.5.3 Пошук з обмеженнями в глибині	39
2.5.4 Пошук в глибину з ітеративним заглибленням	40
2.5.5 Двонаправлений пошук.....	43
2.6 Запобігання формування станів, що повторюються	46
2.7 Евристичні алгоритми, Інформативний пошук	51
2.7.1 Жадібний пошук по першому найкращому співпадінню	53
2.7.2 Пошук A * мінімізація сумарної оцінки вартості рішення	56
2.8 Евристичний пошук з обмеженням обсягу пам'яті	59
2.8.1 Рекурсивний пошук за першим найкращим збігом	60

2.9 Навчання кращим способам пошуку	65
2.10 Евристичні функції.....	66
2.10.1 Складання допустимих евристичних функцій.....	69
2.10.2 Вивчення евристичних функцій на основі досвіду	74
2.11 Алгоритми локального пошуку і задачі оптимізації	75
2.11.1 Пошук зі сходженням до вершини	77
2.11.2 Пошук з емуляцією відпалу	83
2.11.3 Локальний променевий пошук	85
2.12 Задачі виконання обмежень	87
2.12.1 Задачі виконання обмежень	88
2.12.2 Застосування пошуку з поверненням для розв'язання задач CSP	94
2.12.3 Упорядкування змінних та значень	99
2.12.4 Поширення інформації за допомогою обмежень	101
2.12.5 Попередня перевірка	101
2.12.6 Поширення обмежень	102
2.12.7 Обробка спеціальних обмежень	107
2.12.8 Інтелектуальний пошук з поверненнями: пошук в зворотному напрямку	109
3 ТЕМА 3 СКЛАДНІ СТРУКТУРИ ДАНИХ	113
3.1 Індексні файли	113
3.1.1 Файли з щільним індексом, або індексного-прямі файли	115
3.1.2 Файли с нещільним індексом, или індексно-послідовні файли	119
3.2 АВЛ-дерево	122
3.3 Червоно-чорні дерева.....	138
3.4 В-дерево	153

4 ТЕМА 4 СУЧАСНІ ЕВРИСТИЧНІ АЛГОРИТМИ.....	161
4.1 Генетичні алгоритми	163
4.1.1 Еволюційна теорія	163
4.1.2 Природний відбір і генетичне спадкування	164
4.1.3 Задачі оптимізації	164
4.1.4 Робота генетичного алгоритму	166
4.1.5 Представлення генетичної інформації	168
4.1.6 Мутація	169
4.1.7 Загальна схема алгоритму	170
4.2 Мурашині алгоритми	174
4.2.1 Концепція мурашиних алгоритмів	175
4.2.2 Узагальнений алгоритм	176
4.2.3 Етапи вирішення задачі за допомогою мурашиних алгоритмів	178
4.2.4 Застосування мурашиних алгоритмів для задачі комівояжера .	179
4.2.5 Локальні правила поведінки мурах	180
4.2.6 Мурашиний алгоритм для задачі комівояжера в псевдокоді	182
4.3 Алгоритм бджолиної колонії	187
4.3.1 Алгоритм бджолиної колонії для розмальовки графа (класичний випадок).	189
4.3.2 ABC-алгоритм.....	192
5 ТЕМА 4 ТЕОРІЯ ІГОР	198
5.1 Мінімаксний алгоритм	207
5.2 Оптимальні рішення в іграх з кількома гравцями	208
5.3 Алгоритм альфа-бета-відсікань	210
5.4 Неідеальні рішення, що приймаються в реальному часі.....	215
5.5 Ігри, які включають елемент випадковості.....	223

5.6 Ігри з неповною інформацією.....	228
5.6.1 Карткові ігри.....	229
6 ДОДАТОК 1 – СПИСКОВІ СТРУКТУРИ ДАНИХ	255
6.1 Стек	257
6.2 Черга.....	258
6.3 Двостороння черга.....	259
6.4 Черга з пріоритетами.....	260
6.5 Зв'язаний список	261
6.5.1 insert або push - вставка нового елемента,.....	262
6.5.2 search – пошук елемента,	263
6.5.3 delete – видалення елемента,.....	263
6.5.4 обернення списку,	264
6.5.5 пошук циклу	265
7 ДОДАТОК 2 – ДЕРЕВОВИДНІ СТРУКТУРИ ДАНИХ	266
7.1 Основні визначення.....	266
7.2 Бінарні дерева	268
7.3 Обхід дерева	275
7.4 Бінарні дерева пошуку	278
7.5 Збалансовані дерева	283
7.6 Купи (піраміди)	285
7.6.1 Двійкова купа.....	286
8 ДОДАТОК 3 – АЛГОРИТМИ ПОШУКУ У СТРУКТУРАХ ДАНИХ....	292
8.1 Класичні алгоритми пошуку.....	292
8.1.1 Пошук в неупорядкованих множинах даних	292
8.1.2 Пошук в упорядкованих множинах даних	293
8.2 Хешування	303

8.2.1 Таблиці з прямою адресацією	304
8.2.2 Хеш-таблиці.....	305
8.2.3 Уникнення колізій за допомогою ланцюгів	307
8.2.4 Уникнення колізій за допомогою відкритої адресації.....	311
8.2.5 Хеш-функції.....	318

1 ТЕМА 1 – АЛГОРИТМИ СОРТУВАННЯ

1.1 Алгоритми зовнішнього сортування

Зовнішнє сортування - сортування даних, розташованих на периферійних пристроях, які не поміщаються в оперативну пам'ять, тобто коли застосувати одне з внутрішніх сортувань неможливо.

Варто відзначити, що внутрішнє сортування значно ефективніше зовнішнього, так як на звернення до оперативної пам'яті витрачається набагато менше часу, ніж до магнітних дисків.

Найбільш часто зовнішнє сортування використовується в СУБД. Основним поняттям при використанні зовнішнього сортування є поняття відрізка. Відрізком довжини K є послідовність записів $A_i, A_{i+1}, \dots, A_{i+k}$, що в ній всі записи впорядковані по деякому ключу.

Максимальна кількість відрізків у файлі N (всі елементи не впорядковані).

Мінімальна кількість відрізків 1 (всі елементи впорядковані).

Наприклад, в деякому файлі А є одновимірний масив:

12 35 65 0 24 36 3 5 84 90 6 2 30

Поділимо масив на відрізки:

12 35 65 | 0 24 36 | 3 5 84 90 | 6 | 2 30

Можна сказати, що масив у файлі А складається з 5 відрізків.

Наприклад, у деякому файлі В є одновимірний масив:

1 2 3 4 5 6 7 8 9 10

Поділимо масив на відрізки:

| 1 2 3 4 5 6 7 8 9 10 |

Можна сказати, що масив у файлі В складається з 1 відрізка.

Ідея більшості методів полягає в розділенні даних на ряд послідовностей, що поміщаються в оперативну пам'ять. Далі застосовується один з методів внутрішнього сортування, після чого послідовності зливаються. Чим більший

обсяг оперативної пам'яті, тим довші будуть послідовності і, отже, тим меншою виявиться їх кількість, що збільшить швидкість сортування.

Якщо ж обсяг оперативної пам'яті малий, то можна розділити вихідні дані на кілька послідовностей, після чого безпосередньо використовувати процедуру злиття.

Основні методи:

- Пряме злиття.
- Природне злиття.
- Збалансоване багатошляхове злиття.
- Багатофазне сортування(Фібоначчі).

1.1.1 Пряме злиття

Почнемо з того, як можна використовувати в якості методу зовнішнього сортування алгоритм простого злиття. Припустимо, що є послідовний файл А, що складається із записів a_1, a_2, \dots, a_n (для простоти припустимо, що n являє собою степінь числа 2). Будемо вважати, що кожен запис складається рівно з одного елемента, що представляє собою ключ сортування. Для сортування використовуються два допоміжні файли В і С (розмір кожного з них буде $n/2$).

Сортування складається з послідовності кроків, в кожному з яких виконується розподіл вмісту файла А у файли В і С, а потім злиття файлів В і С у файл А. На першому кроці для розподілу послідовно зчитується файл А, і записи a_1, a_3, \dots, a_{n-1} пишуться у файл В, а записи a_2, a_4, \dots, a_n - у файл С (початковий розподіл). Початкове злиття проводиться над парами $(a_1, a_2), (a_3, a_4), \dots, (a_{n-1}, a_n)$, і результат записується у файл А.

На другому кроці знову послідовно зчитується файл А, і в файл В записуються послідовні пари з непарними номерами, а в файл С - з парними. При злитті утворюються і пишуться в файл А впорядковані четвірки записів. І так далі. Перед виконанням останнього кроку, файл А міститиме дві впорядковані підпослідовності розміром $n/2$ кожна. При розподілі перша з них

потрапить у файл В, а друга - у файл С. після злиття файл А міститиме повністю впорядковану послідовність записів.

У таблиці показано приклад зовнішнього сортування простим злиттям:

Початковий стан файлу А = {8, 23, 5, 65, 44, 33, 1, 6}.

№ ітерації	Файл В	Файл С	Файл А
1	{8, 5, 44, 1}	{23, 65, 33, 6}	{8, 23, 5, 65, 33, 44, 1, 6}
2	{8, 23, 33, 44}	{5, 65, 1, 6}	{5, 8, 23, 65, 1, 6, 33, 44}
3	{5, 8, 23, 65}	{1, 6, 33, 44}	{1, 5, 6, 8, 23, 33, 44, 65}

Для виконання зовнішнього сортування методом прямого злиття в основній пам'яті потрібно розташувати всього лише дві змінні - для розміщення чергових записів з файлів В і С. файли А, В і С будуть $O(\log n)$ раз прочитані і стільки ж разів записані.

1.1.2 Природне злиття

При використанні методу прямого злиття не береться до уваги те, що вихідний файл може бути частково відсортованим, тобто містити впорядковані підпослідовності записів. Серією називається підпослідовність записів a_i, a_{i+1}, \dots, a_j така, що $a_k \leq a_{k+1}$ для всіх $i \leq k < j$, $a_i < a_{i-1}$ і $a_j > a_{j+1}$. Метод природного злиття ґрунтуються на розпізнаванні серій при розподілі та їх використанні при подальшому злитті.

Як і в разі прямого злиття, сортування виконується за кілька кроків, в кожному з яких спочатку виконується розподіл файлу А по файлах В і С, а потім злиття В і С в файл А. при розподілі розпізнається перша серія записів і переписується в файл В, друга - в файл С і т. д. При злитті перша серія записів файлу В зливається з першою серією файлу С, друга серія В з другою серією С і т. д. Якщо перегляд одного файла закінчується раніше, ніж перегляд іншого (через різне число серій), то залишок нерозглянутого файла цілком копіюється в кінець файла А. Процес завершується, коли у файла А залишається тільки одна серія. Приклад сортування файла показаний на рисунках:

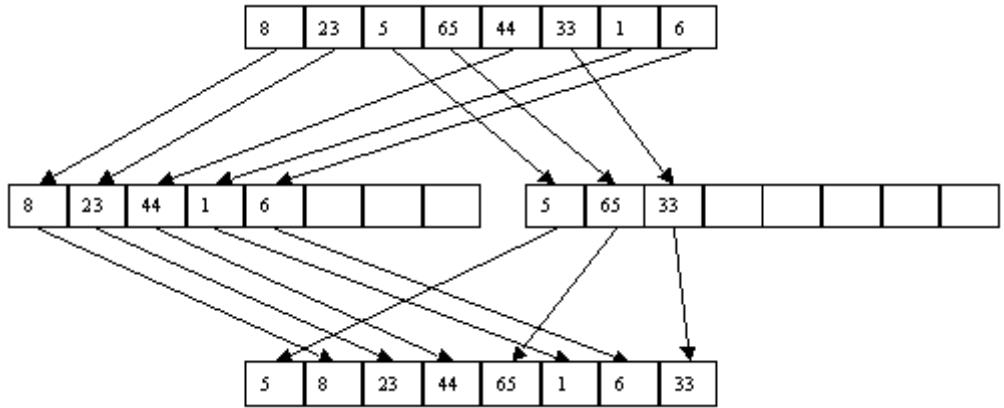


Рисунок 1.1 – Перший крок

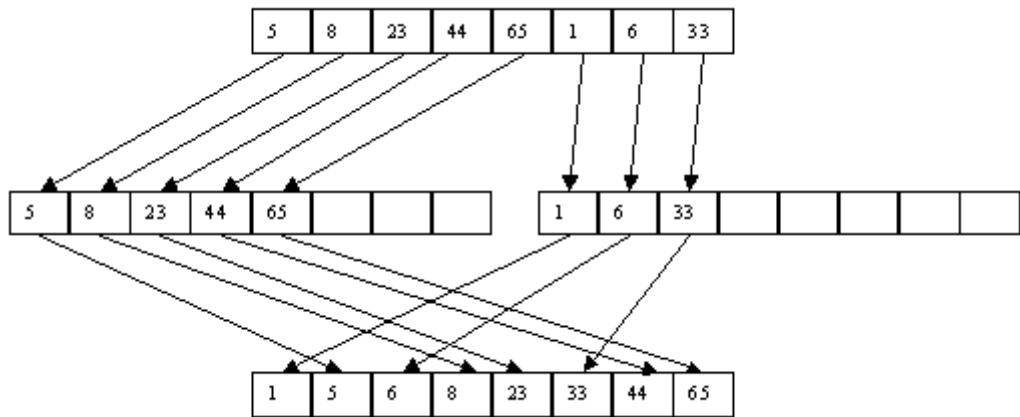


Рисунок 1.2 – Другий крок

Очевидно, що число читань / перезаписів файлів при використанні цього методу буде не гірше, ніж при застосуванні методу прямого злиття, а в середньому - краще. З іншого боку, збільшується число порівнянь за рахунок тих, які потрібні для розпізнавання кінців серій. Крім того, оскільки довжина серій може бути довільною, то максимальний розмір файлів В і С може бути близький до розміру файлу А.

1.1.3 Збалансоване багатошляхове злиття

В основі методу зовнішнього сортування збалансованим багатошляховим злиттям лежить розподіл серій вихідного файлу по m допоміжним файлам B_1, B_2, \dots, B_m і їх злиття в m допоміжних файлів C_1, C_2, \dots, C_m . На наступному кроці проводиться злиття файлів C_1, C_2, \dots, C_m в файли B_1, B_2, \dots, B_m и т.д., поки у B_1 чи C_1 не утвориться одна серія.

Багатофазне злиття є природним розвитком ідеї звичайного злиття.

Приклад тришляхового злиття

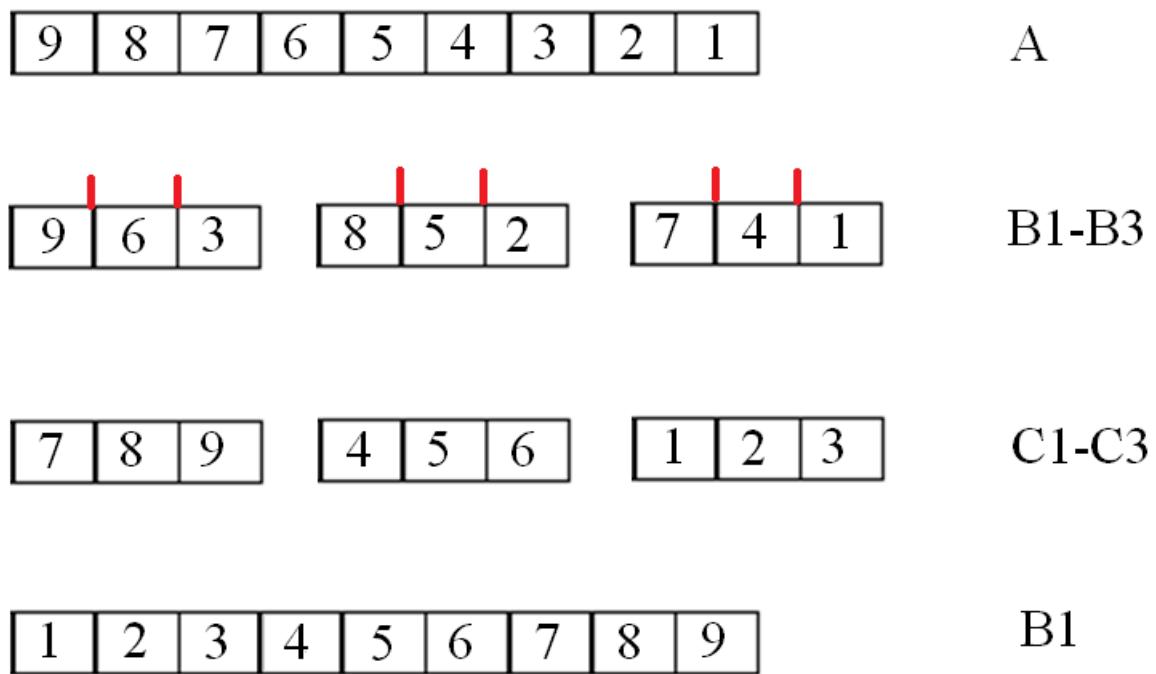
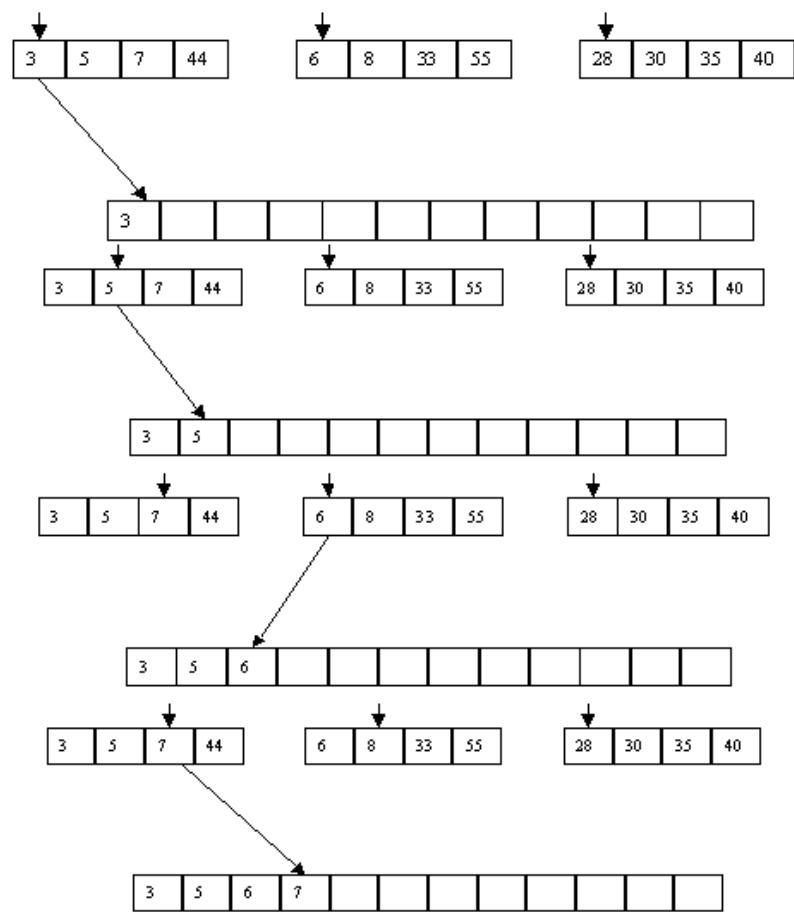
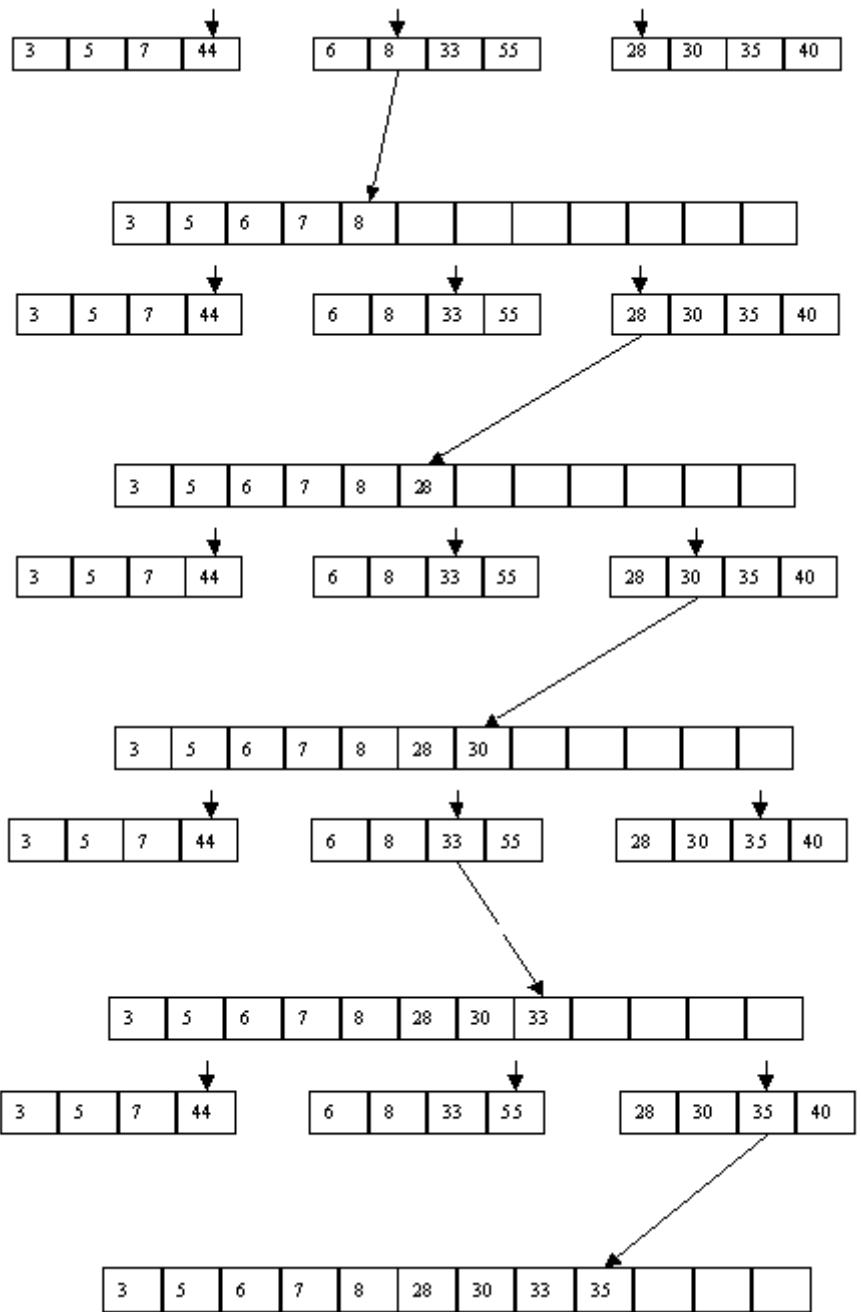


Рисунок 1.3

Приклад тришляхового злиття 2 показаний на малюнку:





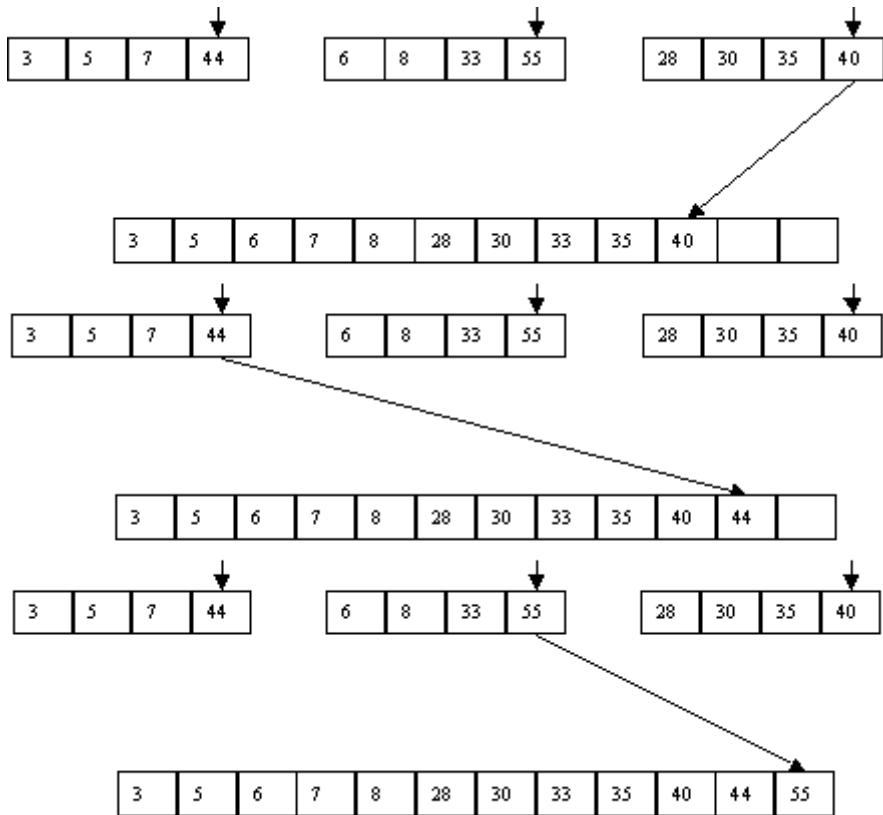


Рисунок 1.4

На рисунку 1.4 показано простий приклад застосування сортування багатошляховим злиттям. Він занадто простий, щоб продемонструвати кілька кроків виконання алгоритму, проте достатній в якості ілюстрації загальної ідеї методу. Зауважимо, що, як показує цей приклад, у міру збільшення довжини серій допоміжні файли з великими номерами (починаючи з номера n) перестають використовуватися, оскільки їм "не дістається" жодної серії. Перевагою сортування збалансованим багатошляховим злиттям є те, що число проходів алгоритму оцінюється як $O(\log n)$ (n - число записів у вихідному файлі), де логарифм береться по основі n . Порядок числа копіювань записів - $O(\log n)$. Звичайно, число порівнянь не буде менше, ніж при застосуванні методу простого злиття.

1.1.4 Багатофазне сортування

При використанні розглянутого вище методу збалансованого багатошляхового зовнішнього сортування на кожному кроці приблизно половина допоміжних файлів використовується для введення даних і приблизно стільки ж для виведення серій, що зливаються. Ідея багатофазного сортування

полягає в тому, що з наявних m допоміжних файлів ($m-1$) файл служить для введення послідовностей, що зливаються, а один - для виведення утворених серій. Як тільки один з файлів введення стає порожнім, його починають використовувати для виведення серій, одержуваних при злитті серій нового набору ($m-1$) файлів. Таким чином, є перший крок, при якому серії вихідного файлу розподіляються по $m-1$ допоміжному файлу, а потім виконується багатошляхове злиття серій з ($m-1$) файлу, поки в одному з них не утворюється одна серія.

Очевидно, що при довільному початковому розподілі серій по допоміжних файлах, алгоритм може не зйтися, оскільки в єдиному непорожньому файлі буде існувати більше, ніж одна серія. Припустимо, наприклад, що використовується три файли B_1 , B_2 і B_3 , і при початковому розподілі в файл B_1 поміщені 10 серій, а в файл B_2 - 6. При злитті B_1 і B_2 до моменту, коли ми дійдемо до кінця B_2 , в B_1 залишиться 4 серії, А в B_3 потраплять 6 серій. Продовжиться злиття B_1 і B_3 , і при завершенні перегляду B_1 в B_2 будуть міститися 4 серії, А в B_3 залишиться 2 серії. Після злиття B_2 і B_3 в кожному з файлів B_1 і B_2 буде міститися по 2 серії, які будуть злиті і утворять 2 серії в B_3 при тому, що B_1 і B_2 - порожні. Тим самим, алгоритм не зійшовся:

Число серій у файлі B_1	Число серій у файлі B_2	Число серій у файлі B_3
10	6	0
4	0	6
0	4	2
2	2	0
0	0	2

Яким же має бути початковий розподіл серій, щоб алгоритм трифазного сортування благополучно завершував роботу і виконувався максимально ефективно? Для цього розглянемо роботу алгоритму в зворотному порядку, починаючи від бажаного кінцевого стану допоміжних файлів. Підходить будь-яка комбінація кінцевого числа серій в файлах В1, В2 і В3 з (1,0,0), (0,1,0) и (0,0,1).

Для визначеності виберемо першу комбінацію. Для того, щоб вона склалася, необхідно, щоб на безпосередньо попередньому етапі злиття існував розподіл серій (0,1,1). Щоб отримати такий розподіл, необхідно, щоб на безпосередньо попередньому етапі злиття розподіл виглядав як (1,2,0) або (1,0,2). Знову для визначеності зупинимося на першому варіанті. Щоб його отримати, на попередньому етапі годилися б такі розподіли: (3,0,2) і (0,3,1). Але другий варіант гірше, оскільки він приводиться до злиття тільки однієї серії з файлів В2 і В3, в той час як при наявності першого варіанту розподілу будуть злиті дві серії з файлів В1 і В3. Побажанням до попереднього етапу була б наявність розподілу (0,3,5), ще раніше - (5,0,8), ще раніше - (13,8,0) і т. д.

Цей розгляд показує, що метод трифазного зовнішнього сортування дає бажаний результат і працює максимально ефективно (на кожному етапі зливається максимальне число серій), якщо початковий розподіл серій між допоміжними файлами описується сусідніми числами Фібоначчі. Нагадаємо, що послідовність чисел Фібоначчі починається з 0, 1, а кожне наступне число утворюється як сума двох попередніх:

$$(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots)$$

Аналогічні міркування показують, що в загальному вигляді при використанні m допоміжних файлів умовою успішного завершення і ефективної роботи методу багатофазного зовнішнього сортування є те, щоб початковий розподіл серій між $m-1$ файлами описувалося сумами сусідніх ($m-1$), ($m-2$), ..., 1 чисел Фібоначчі порядку $m-2$. Послідовність чисел Фібоначчі порядку p починається з p нулів, $(p+1)$ -й елемент дорівнює 1, а кожен

наступний дорівнює сумі попередніх $p+1$ елементів. Нижче показано початок послідовності чисел Фібоначчі порядку 4:

$$(0, 0, 0, 0, 1, 1, 2, 4, 8, 16, 31, 61, \dots)$$

Якщо розподіл заснований на числі Фібоначчі f_i , то мінімальне число серій у допоміжних файлах дорівнюватиме f_i , а максимальне - $f(i+1)$. Тому після виконання злиття ми отримаємо максимальне число серій- f_i , а мінімальне - $f(i-1)$. На кожному етапі буде виконуватися максимально можливе число злиттів, і процес зідеться до наявності всього однієї серії.

Оскільки число серій у вхідному файлі може не забезпечувати можливість такого розподілу серій, застосовується метод додавання порожніх серій, які в подальшому якомога більш рівномірного розподіляються між проміжними файлами і розпізнаються при наступних злиттях. Зрозуміло, що чим менше таких порожніх серій, тобто чим біжче число початкових серій до вимог Фібоначчі, тим ефективніше працює алгоритм.

Приклад: є два файли з числом серій 13 і 8. Результатуючий файл матиме 21 серію даних.

Фаза	Файл ₁	Файл ₂	Файл ₃	Пояснення
1	13	8	0	8 блоків даних з файла ₂ об'єднуються з 8-ма блоками даних із файла ₁ і записуються у файл ₃ .
2	5	0	8	У файлі ₁ залишається 5 блоків, у файлі ₂ даних для об'єднання не залишається.
3	0	5	3	5 блоків даних із файла ₁ об'єднуються з 5-ма блоками даних з файла ₃ і записуються у файл ₂ . У файлі ₃ залишається ще 3 блока даних.
4	3	2	0	3 блока даних з файла ₃ об'єднуються з 3-ма блоками даних з файла ₂ і записуються у файл ₁ . У файлі ₂ залишається ще 2 блока.
5	1	0	2	2 блока із файла ₂ об'єднуються з 2-ма блоками даних з файла ₁ і записуються в файл ₃ . У файлі ₃ залишається один блок.
6	0	1	1	1 блок із файла ₁ об'єднується з одним блоком з файла ₃ і записується в файл ₂ . У файлі ₃ залишається 1 блок.
7	1	0	0	1 блок з файла ₂ об'єднується з 1 блоком із файла ₃ і записується в файл ₁ . У файлі ₂ і файлі ₃ даних немає.

1.1.5 Покращення ефективності зовнішнього сортування за рахунок використання оперативної пам'яті

Чим довші серії містить файл перед початком застосування зовнішнього сортування, тим менше буде потрібно злиттів і тим швидше закінчиться сортування. Тому до початку застосування будь-якого з методів зовнішнього сортування, заснованих на застосуванні серій, початковий файл частинами зчитується в оперативну пам'ять, до кожної частини застосовується один з найбільш ефективних алгоритмів внутрішнього сортування і відсортовані частини, що утворюють серії, записуються в новий файл.

Крім того, при виконанні розподілів і злиттів використовується буферизація блоків файлу в оперативній пам'яті. Можливий виграш у продуктивності залежить від наявності достатнього числа буферів достатнього розміру. Досить часто при використанні буферизації виникає необхідність модифікувати стандартні операції розділення та злиття описані в алгоритмах вище.

Відображення файлу в пам'ять – це спосіб роботи з файлами в деяких операційних системах, при якому всьому файлу або певній безперервній його частині ставиться у відповідність певну ділянку пам'яті (діапазон адрес оперативної пам'яті). При цьому читання даних із цих адрес фактично призводить до читання даних з відображеного файла, а запис даних за цими адресами призводить до запису цих даних у файл.

2 ТЕМА 2 АЛГОРИТМИ ПОШУКУ

Пошук в рамках штучного інтелекту (у просторі станів) – це процес переміщення з початкового стану в цільове через проміжні.

В цьому розділі перераховані деякі з найбільш відомих прикладів вирішення задач, які діляться на два типи – спрощені та реальні задачі. Спрощена задача призначена для ілюстрації або перевірки різноманітних методів вирішення задач. Їй може бути дано короткий, точний опис. Це означає, що така задача може легко використовуватися різними дослідниками для порівняння продуктивності алгоритмів. Реальною задачею називається така задача, рішення якої дійсно потрібно людям. Як правило, такі задачі не мають єдиного прийнятного для всіх опису, але ми спробуємо показати, як зазвичай виглядають їх формуллювання.

2.1 Спрощені задачі

Як перший приклад розглянемо світ пилососа. Діяльність в цьому світі можна сформулювати в якості задачі, як описано нижче.

- **Стан.** Агент (пилосос) знаходиться в одному з двох місцезнаходжень, в кожному з яких може бути або не бути присутнім сміття. Тому існує $2 \times 2^2 = 8$ можливих станів світу.
- **Початковий стан** В якості початкового стану може бути призначено будь-який стан.
- **Функція визначення спадкоємця.** Ця функція виробляє допустимі стани, які є спадкоємцями спроб виконання трьох дій (*Left*, *Right* i *Suck*). Повний простір станів показано на рисунку 2.1.

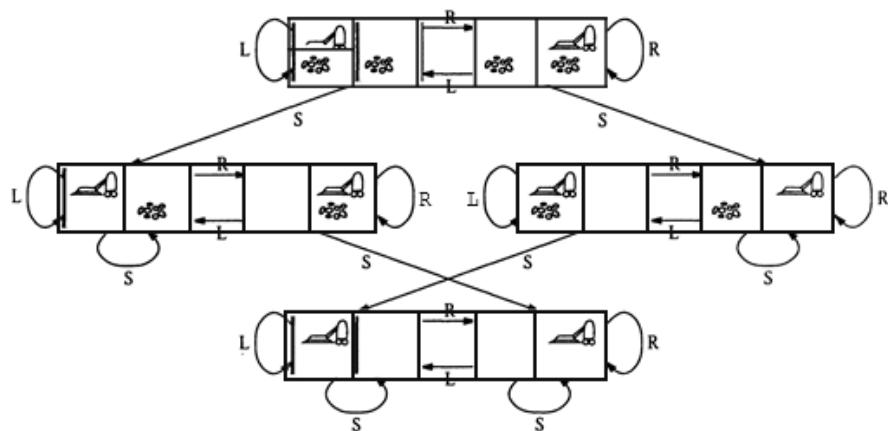


Рисунок 2.1 – Простір станів для світу пилососа. **Дуги** позначають дії.
L=Left, R=Right, S=Suck

- **Перевірка цілі.** Ця перевірка зводиться до визначення того, чи є чистими всі квадрати.
- **Вартість шляху.** Вартість кожного етапу дорівнює 1, тому вартість шляху являє собою кількість етапів в цьому шляху.

У порівнянні із завданням реального світу ця спрощена задача характеризується розрізняваними місцезнаходженнями, можливістю визначати наявність сміття, надійним очищеннем, а також збереженням досягнутого стану після очищення. Необхідно враховувати, що стан визначається як місцезнаходженням агента, так і наявністю сміття. У більшому середовищі з **n** місцезнаходженнями існує **n^*2^n** станів.

Задача гри у вісім, екземпляр якої показаний на рисунку 2.2, складається з дошки 3x3 з вісімома пронумерованими фішками і з однією порожньою ділянкою. Фішка, суміжна з порожньою ділянкою, може бути перенесена на цю ділянку. Потрібно досягти зазначеного цільового стану, подібного до того, який показано в правій частині малюнка. Стандартне формулювання цього завдання наведена нижче. .

- **Стан.** Опис стану визначає місцезнаходження кожної з цих восьми фішок і порожньої ділянки на одному з дев'яти квадратів.
- **Початковий стан.** Початковим може бути визначено будь-який стан. Необхідно відзначити, що будь-яка задана мета може бути досягнута точно з половини можливих початкових станів (інші не мають розв'язку).
- **Функція визначення спадкоємця.** Ця функція формує допустимі стани, які є наслідком спроб виконання вказаних чотирьох дій (теоретично можливих ходів *Left, Right, Up i Down*).
- **Перевірка цілі.** Вона дозволяє визначити, чи відповідає даний стан цільовій конфігурації, показаній на рисунку 2.2. (Можливі також інші цільові конфігурації.)
- **Вартість шляху.** Вартість кожного етапу дорівнює 1, тому вартість шляху являє собою кількість етапів в цьому шляху.

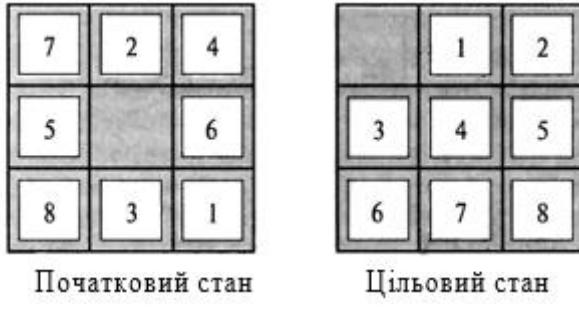


Рисунок 2.2 – Типовий екземпляр задачі гри у вісім

Задача гри у вісім відноситься до сімейства задач з **ковзаючими фішками**, які часто використовуються в штучному інтелекті для перевірки нових алгоритмів пошуку. Відомо, що цей загальний клас задач є NP-повним, тому навряд чи можна сподіватися знайти методи, які в найгіршому випадку були б набагато краще в порівнянні з алгоритмами пошуку, описаними в цьому та наступних розділах. Задача гри у вісім має $9! / 2 = 181440$ досяжних станів і легко вирішується. Задача гри в п'ятнадцять (на дощі 4x4) має близько 1,3 трильйона станів, і випадково вибрані її екземпляри можуть бути вирішені оптимальним чином за кілька мілісекунд за допомогою найкращих алгоритмів пошуку.

Мета задачі з **вісімома ферзями** заключається в розміщенні восьми ферзів на шахівниці таким чином, щоб жоден ферзь не нападав на будь-якого іншого. (Ферзь атакує будь-яку фігуру, що знаходиться на одній і тій же з ним горизонталі, верикалі або діагоналі.) На рисунку 2.3 показана невдала спроба пошуку рішення: ферзь, що знаходиться на крайній правій верикалі, атакований ферзем, що знаходиться угорі ліворуч.

Незважаючи на те, що існують ефективні спеціалізовані алгоритми вирішення цієї задачі і всього сімейства задач з n ферзями, вона як і раніше залишається цікавою експериментальною задачею для алгоритмів пошуку. Для неї застосовуються формулювання двох основних типів. В **інкрементному формулуванні** передбачається використання операторів, які доповнюють опис стану, починаючи з порожнього стану; для задачі з вісімома ферзями це означає, що кожна дія приводить до додавання до цього стану ще одного ферзя.

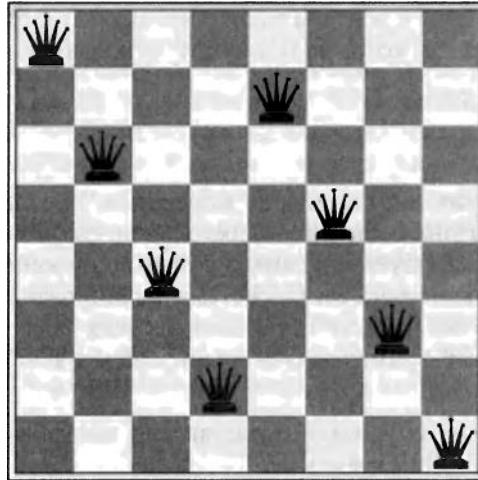


Рисунок 2.3 – Майже готове рішення задачі з вісъмома ферзями

Формулювання повного стану починається з установки на дошку всіх восьми ферзів та передбачає їх подальше переміщення. В тому і іншому випадку вартість шляху не представляє інтересу, оскільки важливо лише досягти кінцевого стану.

Перше інкрементне формулювання, яке може застосовуватися при здійсненні спроб вирішення цієї задачі, наведена нижче.

- **Стан.** Станом є будь-яке розташування ферзів на дошці в кількості від 0 до 8.
- **Початковий стан.** Відсутність ферзів на шахівниці.
- **Функція визначення спадкоємця.** Встановлення ферзя на будь-який порожній клітці.
- **Перевірка цілі.** На шахівниці знаходиться вісім ферзів, і жоден з них не атакований.

У цьому формулюванні потрібно перевірити $64 \cdot 63 \cdot \dots \cdot 57 = 3 \cdot 10^{14}$ можливих послідовностей. У кращому формулюванні повинно бути заборонено поміщати ферзя на будь-яку клітину, яка вже атакована, таким чином:

- **Стан.** Станами являються розміщення з n ферзів ($0 \leq n \leq 8$), по одному ферзю в кожній з n вертикалей, при яких жоден ферзь не атакує іншого.
- **Функція визначення спадкоємця.** Установка ферзя на будь-якій клітинці на порожній вертикалі таким чином, щоб він не був атакований будь-яким іншим ферзем.

Це формулювання дозволяє скоротити простір станів задачі з вісъмома ферзями з $3 \cdot 10^{14}$ до 2057, і пошук рішень значно спрощується. З іншого боку, для 100 ферзів первинне формулювання визначає приблизно 10^{400} станів, а

поліпшене формулювання - близько 10^{52} станів. Це – колосальне скорочення, але покращений простір станів все ще завеликий для того, щоб з ним могли впоратися алгоритми, що розглядаються в даному розділі.

2.2 Реальні задачі

Вище вже було показано, як можна визначити **задачу пошуку маршруту** в термінах заданих місцезнаходжень і переходів по зв'язках між ними. Алгоритми пошуку маршруту використовуються в найрізноманітніших застосунках, таких як системи маршрутизації в комп'ютерних мережах, системи планування військових операцій і авіаподорожей. Зазвичай процес визначення таких задач є трудомістким. Розглянемо спрощений приклад задача планування авіаподорожей, який заданий наступним чином.

- **Стан.** Кожний стан представлено місцезнаходженням (наприклад, аеропортом) і поточним часом.
- **Початковий стан.** Задається умовою задачі.
- **Функція визначення спадкоємця.** Ця функція повертає стани, які випливають з виконання будь-якого польоту, зазначеного в розкладі (можливо, з додатковою вказівкою класу і місця), відправлення пізніше порівняно з поточним часом з урахуванням тривалості переміщень в межах самого аеропорту, а також з одного аеропорту в інший.
- **Перевірка цілі.** Чи знаходимося ми в місці призначення до деякого заздалегідь заданому часу?
- **Вартість шляху.** Залежить від вартості квитка, часу очікування, тривалості польоту, митних і імміграційних процедур, комфортності місця, часу доби, типу літака, знижок для постійних клієнтів і т.д.

У комерційних консультаційних системах планування подорожей використовується формулювання задачі такого типу з багатьма додатковими ускладненнями, які потрібні для обліку надзвичайно заплутаних структур визначення плати за проїзд, застосовуваних в авіакомпаніях. Але будь-який досвідчений мандрівник знає, що не всі авіаподорожі проходять згідно з планом. Дійсно якісна система повинна передбачати плани дій в непередбачених ситуаціях (такі як страхувальне резервування квитків на альтернативні рейси) в такій мірі, яка відповідає вартості і ймовірності порушення початкового плану.

Задачі планування обходу тісно пов'язані з задачами пошуку маршруту, але з одним важливим винятком. Розглянемо, наприклад, задачу: "Відвідати кожне місто, показане на рисунку 2.4, щонайменше один раз, почавши і закінчивши подорож в Бухаресті". Як і при пошуку маршруту, дії відповідають поїздкам з одного суміжного міста в інше. Але простір станів є зовсім іншим. Кожен стан має включати не лише поточне місцезнаходження, але і множину міст, які відвідав агент. Тому початковим станом повинно бути "В Бухаресті; відвіданий {Бухарест}", а типовим проміжним станом – "В Васлуй; відвідані {Бухарест, Урзічені, Васлуй}", тоді як перевірка цілі повинна передбачати визначення того, чи знаходиться агент в Бухаресті і чи відвідав він всі 20 міст.

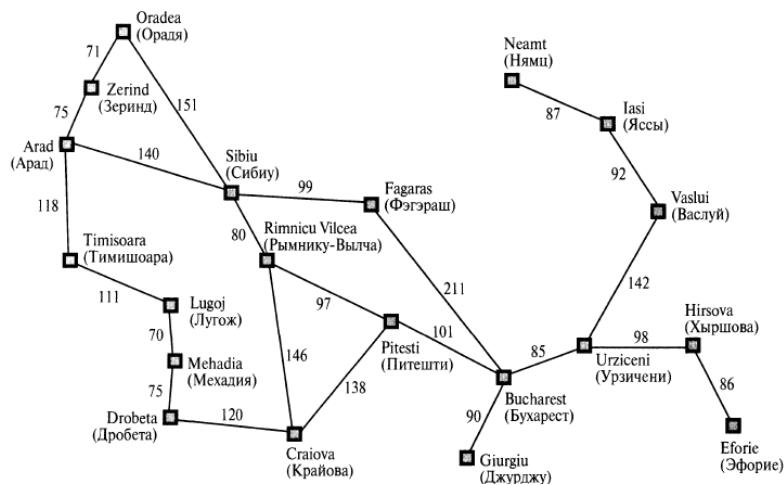


Рисунок 2.4 – Спрощена дорожня карта Румунії

Однією із задач планування обходу є **задача комівояжера** (Traveling Salesperson Problem - TSP), за умовою якої кожне місто повинне бути відвідане лише один раз. Призначення її полягає в тому, щоб знайти найкоротший шлях обходу. Як відомо, ця задача є NP-складною, але на поліпшення можливостей алгоритмів TSP були витрачені колосальні зусилля. Крім планування поїздок комівояжерів, ці алгоритми використовувалися для вирішення таких задач, як планування переміщень автоматичних свердел при розробці друкованих плат і організація роботи засобів постачання в виробничих цехах.

Задача управління навігацією робота являє собою узагальнення описаної вище задачі пошуку маршруту. У цій задачі замість дискретної множини маршрутів розглядається ситуація, в якій робот може переміщатися в безперервному просторі з нескінченою (в принципі) множиною можливих дій

і станів. Якщо потрібно забезпечити циклічне переміщення робота по плоскій поверхні, то простір фактично може розглядатися як двовимірний, а якщо робот обладнаний верхніми і нижніми кінцівками або колесами, якими також необхідно управляти, то простір пошуку стає багатовимірним. Навіть для того щоб зробити цей простір пошуку кінцевим, потрібні досить розвинені методи. Початкова складність завдання погіршується тим, що при управлінні реальними роботами необхідно враховувати помилки в показаннях датчиків, а також відхилення в роботі рухових засобів управління.

Вирішення задачі **автоматичного впорядкування збірки** складних об'єктів роботом було вперше продемонстровано на прикладі робота Freddy. З тих пір прогрес в цій області відбувається повільно, але впевнено, і в даний час стала економічно вигідною збірка таких неординарних об'єктів, як електродвигуни. У задачах збірки мета полягає у визначенні послідовності, в якій повинні бути зібрані деталі деякого об'єкту. Якщо обрана неправильна послідовність, то в подальшому не можна буде знайти спосіб додавання деякої деталі до цієї послідовності без скасування певної частини вже виконаної роботи. Перевірка можливості виконання деякого етапу в послідовності являє собою складну геометричну задачу пошуку, тісно пов'язану з задачею навігації робота. Тому одним з дорогих етапів вирішення задачі впорядкування збірки є формування допустимих спадкоємців. Будь-який практично застосовний алгоритм повинен запобігти необхідності пошуку в усьому просторі станів, за винятком крихітної його частини. Ще однією важливою задачею збірки є **проектування молекули білка**, мета якої полягає у визначенні послідовності амінокислот, здатних скластися в тривимірний білок з потрібними властивостями, призначений для лікування деяких захворювань.

В останні роки зросла потреба в створенні програмних роботів, які здійснюють пошук в **Internet**, знаходячи відповіді на питання, відшукуючи необхідну інформацію або здійснюючи торгові операції. Це - хороше застосування для методів пошуку, оскільки **Internet** легко уявити концептуально у вигляді графа, що складається з вузлів (сторінок), з'єднаних за допомогою посилань.

2.3 Пошук рішень

Сформулювавши певні задачі, необхідно знайти їх рішення. Така мета досягається за допомогою пошуку в просторі станів. У цьому розділі розглядаються методи пошуку, в яких використовується явно задане **дерево пошуку**, що створюється за допомогою початкового стану і функції визначення спадкоємця, які спільно задають простір станів. Замість дерева пошуку може застосовуватися граф пошуку, якщо один і той же стан може бути досягнутий за допомогою багатьох шляхів.

На рисунку 2.5 показані деякі розширення дерева пошуку, потрібного для визначення маршруту від Арада до Бухареста. Коренем цього дерева пошуку є **пошуковий вузол**, відповідний початкового стану, In (Arad). Перший етап полягає в перевірці того, чи є цей стан цільовим. Безумовно, що він не є таким, але необхідно передбачити відповідну перевірку, щоб можна було вирішувати задачі, що містять в собі готове рішення, такі як "почавши подорож з міста Арад, прибути в місто Арад". А в даному випадку поточний стан не є цільовим, тому необхідно розглянути деякі інші стани. Такий етап здійснюється шляхом **розгортання** поточного стану, тобто застосування функції визначення спадкоємця до поточного стану для **формування** в результаті цього нової множини станів. В даному випадку будуть отримані три нові стани: In (Sibiu), In (Oradea) та In (Zerind). Тепер необхідно визначити, який з цих трьох варіантів слід розглядати далі.

В цьому і полягає суть пошуку - поки що перевірити один варіант і відкласти інші в сторону, на випадок, якщо перший варіант не приведе до рішення. Припустимо, що спочатку обрано місто Сібіу. Проведемо перевірку для визначення того, чи відповідає він цільовому стану (не відповідає), а потім розгорнемо вузол Sibiu для отримання станів In (Arad), In (Fagaras), In (Oradea) та In (Rimnicu Vilcea). Після цього можна вибрати будь-який з цих чотирьох станів або повернутися і вибрати вузол Timisoara чи Zerind. Необхідно знову і знову вибирати, перевіряти і розгорнати вузли до тих пір, поки не буде знайдено рішення або не залишиться більше станів, які можна було б розгорнути. Порядок, в якому відбувається розгортання

станів, визначається стратегією пошуку. Загальний алгоритм пошуку в дереві неформально представлений в лістингу 1.

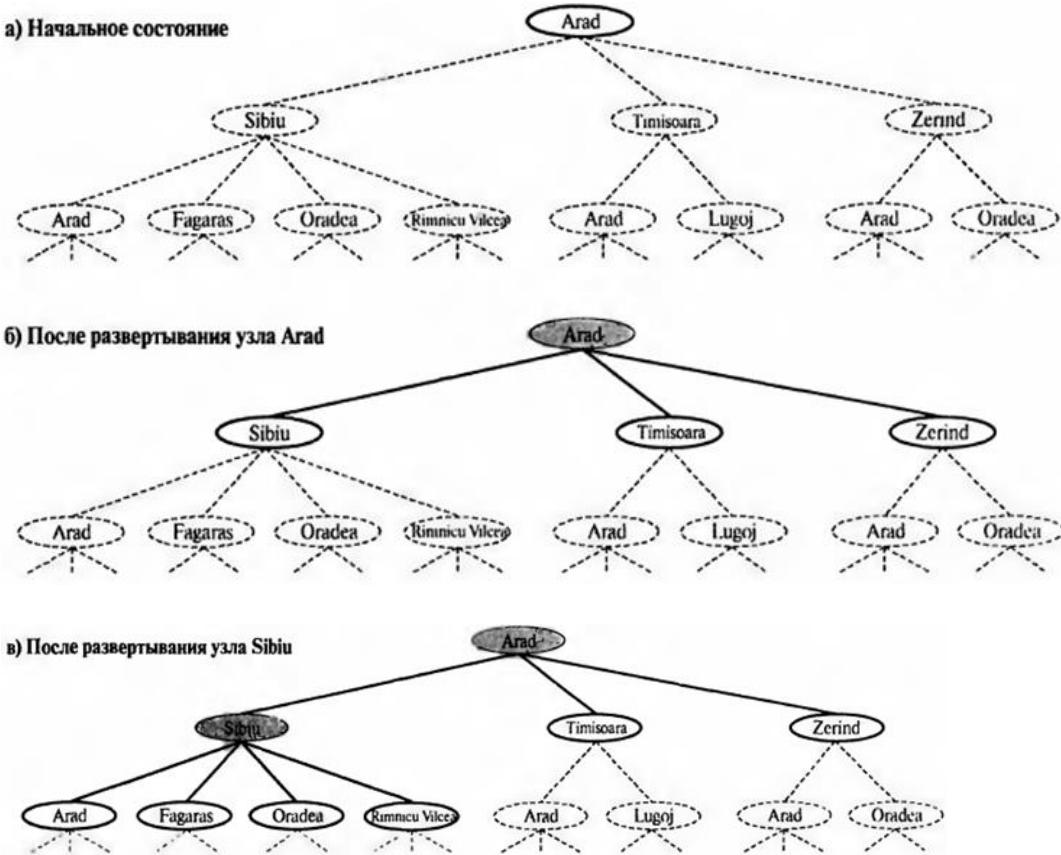


Рисунок 2.5 – Частково розгорнуті дерева пошуку, призначені для визначення маршруту від Арада до Бухареста. Розгорнуті вузли затінені; вузли, які були сформовані, але ще не розгорнуті, виділені напівжирним контуром; вузли, які ще не були сформовані, позначені тонкими штриховими лініями

Лістинг 1. Неформальний опис загального алгоритму пошуку в дереві

```

function Tree-Search(problem, strategy) returns рішення solution або ідентифікатор
неудачі failure задачі problem
    ініціалізувати дерево пошуку з використанням початкового стану
    loop do             невдачі
        if немає кандидатів на розгортання then return ідентифікатор
failure                     невдачі
        if цей вузол містить цільовий стан
            then return відповідне рішення solution стратегії strategy
        else розгорнути цей вузол і додати отримані вузли до дерева пошуку
    Необхідно враховувати відмінність між простором станів і деревом
    пошуку. У просторі станів для заданого пошуку маршруту є тільки 20 станів, по
  
```

одному для кожного міста. Але кількість шляхів в цьому просторі станів являється умовно нескінченим, тому дерево пошуку має нескінченну кількість вузлів. Наприклад, першими трьома шляхами будь-якої нескінченної послідовності шляхів є маршрути Арад-Сібіу, Арад-Сібіу-Арад, Арад-Сібіу-Арад-Сібіу. (Безумовно, якісний алгоритм пошуку повинен виключати можливість формування таких повторюваних шляхів.)

Існує безліч способів подання вузлів, але тут передбачається, що вузол являє собою структуру даних з п'ятьма компонентами, які описані нижче.

- **State.** Стан в просторі станів, якому відповідав би даний вузол.
- **Parent-Node.** Вузол в дереві пошуку, що застосовувався для формування даного вузла (батьківський вузол).
- **Action.** Дія, яка була застосована до батьківського вузла для формування даного вузла.
- **Path-Cost.** Вартість шляху (від початкового стану до даного вузла), показаного за допомогою покажчиків батьківських вузлів, яку прийнято позначати як $g(n)$.
- **Depth.** Кількість етапів шляху від початкового стану, що також називають глибиною.

Необхідно враховувати відмінність між вузлами і станами. Вузол - це облікова структура даних, що застосовується для представлення дерева пошуку, а стан відповідає конфігурації світу. Тому вузли лежать на конкретних шляхах, які визначені за допомогою покажчиків Parent-Node, а стани - ні. Крім того, два різних вузла можуть включати один і той же стан світу, якщо цей стан формується з допомогою двох різних шляхів пошуку. Структура даних вузла показана на рисунку 2.6.

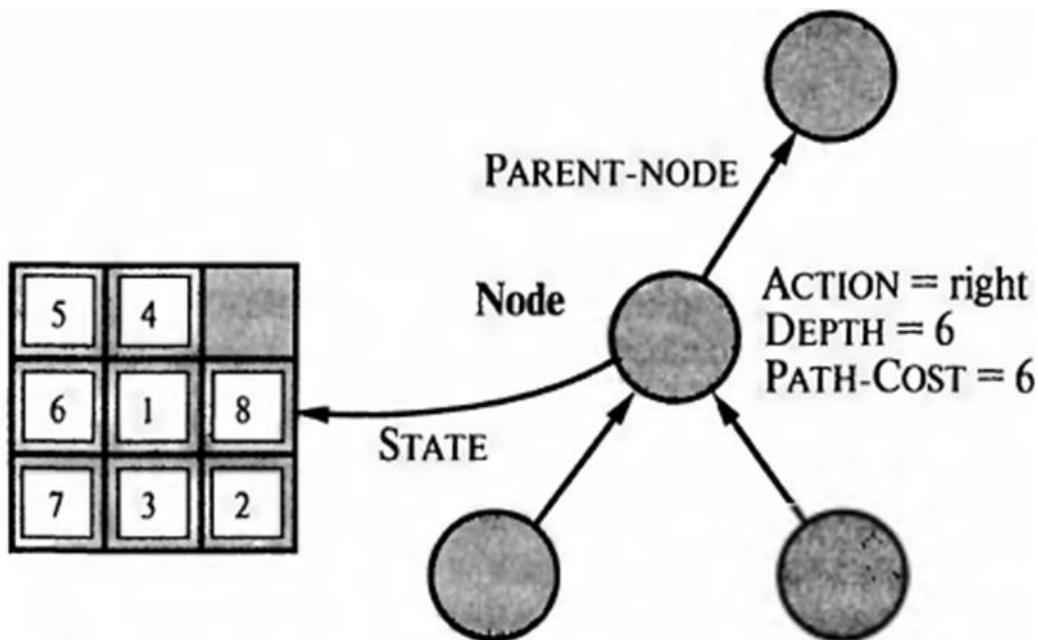


Рисунок 2.6 – Вузли представляють собою структури даних, за допомогою яких формується дерево пошуку. Кожен вузол має батьківський вузол, містить дані про стан і має різні допоміжні поля. Стрілки спрямовані від дочірнього вузла до батьківського

Необхідно також представити колекцію вузлів, які були сформовані, але ще не розгорнуті; така колекція називається **периферією**. Кожен елемент периферії являє собою **листовий вузол**, тобто вузол, який не має наступників в дереві. На рисунку 2.5 периферія кожного дерева складається з вузлів з напівжирними контурами. Найпростішим поданням периферії може служити множина вузлів. Тоді стратегія пошуку повинна бути виражена у вигляді функції, яка вибирає певним чином з цієї множини наступний вузол, що підлягає розгортанню. Хоча даний підхід концептуально є нескладним, він може виявитися дорогим з обчислювальної точки зору, оскільки функцію, передбачену в цій стратегії, можливо, доведеться застосувати до кожного елементу в зазначеній множині для вибору найкращого з них. Тому передбачається, що колекція вузлів реалізована у вигляді **черги**. Операції, що можуть бути застосовані до будь-якої черги:

- `Make-Queue (element , ...)`. Створює чергу з заданим елементом (елементами).
- `Empty?(queue)`. Повертає `True` якщо в черзі більше немає елементів.
- `First(queue)`. Повертає перший елемент черги.
- `Remove-First(queue)`. Повертає елемент `First(queue)` і видаляє його із черги.

- `Insert(element, queue)`. Додає елемент до черги та повертає результиручу чергу.
- `Insert-All(elements, queue)`. Додає множину елементів до черги та повертає результиручу чергу.

За допомогою цих визначень ми можемо записати більш формальну версію загального алгоритму пошуку в дереві, показану в лістингу 2.

Лістинг 2. Загальний алгоритм пошуку в дереві. Слід враховувати, що фактичний параметр `fringe` (периферія) повинен являти собою порожню чергу, а порядок пошуку залежить від типу черги. Функція `Solution` повертає послідовність дій, отриманих шляхом проходження по вказівникам на батьківські вузли в зворотному напрямку, до кореню

```

function Tree-Search(problem, strategy) returns рішення solution або ідентифікатор
недачі failure
    fringe  $\leftarrow$  Insert(Make-Node(Initial-State[problem]), fringe)
    loop do
        if Empty?(fringe) then return ідентифікатор недачі failure
        node  $\leftarrow$  Remove-First(fringe)
        if Goal-Test[problem] застосоване до State[node] завершується успішно
            then return Solution(node)
        fringe  $\leftarrow$  Insert-All(Expand(node, problem), fringe)

function Expand(node, problem) returns множину вузлів successors
    successors  $\leftarrow$  пуста множина
    for each <action, result> in Succesors-Fn[problem](State[node]) do
        s  $\leftarrow$  новий вузол
        State[s]  $\leftarrow$  result
        Parent-Node[s]  $\leftarrow$  node
        Action[s]  $\leftarrow$  action
        Path-Cost[s]  $\leftarrow$  Path-Cost[node] + Step-Cost(node, action, s)
        Depth[s]  $\leftarrow$  Depth[node] + 1
        додати вузол s до множини Successors

```

2.4 Вимірювання продуктивності рішення задачі

Результатом застосування будь-якого алгоритму розв'язання задачі є або невдале завершення, або отримання рішення. (Деякі алгоритми можуть входити в нескінченний цикл і не повернати ніякого результату.) Ми будемо оцінювати продуктивність алгоритму за допомогою чотирьох показників, описаних нижче.

- **Повнота.** Чи гарантує алгоритм виявлення рішення, якщо воно є?

- **Оптимальність.** Чи забезпечує дана стратегія знаходження оптимального рішення?
- **Часова складність.** За якийсь час алгоритм знаходить рішення?
- **Просторова складність.** Який обсяг пам'яті необхідний для здійснення пошуку?

Часова і просторова складність завжди аналізуються з урахуванням певного критерію вимірювання складності задачі. У теоретичній комп'ютерній науці типовим критерієм є розмір графа простору станів, оскільки цей граф розглядається як явно задана структура даних, яка є вхідною для програми пошуку. (Прикладом цього може слугувати карта Румунії.) У штучному інтелекті, де граф представлений неявно за допомогою початкового стану і функції визначення спадкоємця і часто є нескінченим, складність виражається в термінах трьох величин: b - **коєфіцієнт розгалуження** або максимальна кількість спадкоємців будь-якого вузла, d - **глибина** найбільш поверхневого цільового вузла і m - **максимальна довжина** будь-якого шляху в просторі станів.

Часова складність часто вимірюється в термінах кількості вузлів, що створюються в процесі пошуку, а просторова складність - в термінах максимальної кількості вузлів, що зберігаються в пам'яті.

Щоб оцінити ефективність будь-якого алгоритму пошуку, можна розглядати тільки **вартість пошуку**, яка зазвичай залежить від часової складності, але може також включати вираз для оцінки використання пам'яті, або застосовувати сумарну вартість, в якій об'єднується вартість пошуку і вартість шляху знайденого рішення. Для задачі пошуку маршруту від Арада до Бухареста вартість пошуку являє собою кількість часу, витраченого на цей пошук, а вартість рішення виражає загальну довжину шляху в кілометрах. Тому для обчислення сумарної вартості нам доведеться складати кілометри і мілісекунди. Між цими двома одиницями вимірювання не визначений "офіційний курс обміну", але в даному випадку було б обґрунтовано перетворювати кілометри в мілісекунди з використанням оцінки середньої швидкості автомобіля (оскільки для даного агента важливим є саме час). Це

дозволяє даному агенту знайти оптимальну точку компромісу, в якій подальші обчислення для пошуку більш короткого шляху стають непродуктивними.

2.5 Стратегії неінформативного пошуку

У цьому розділі розглядаються п'ять стратегій пошуку, які відомі під назвою **неінформативний пошук** (також називають **сліпим пошуком**). Цей термін означає, що в даних стратегіях не використовується додаткова інформація про стани, крім тієї, що подана в описі задачі. Все, на що вони здатні, - створювати спадкоємців і відрізняти цільовий стан від нецільового. Стратегії, що дозволяють виявити, чи є один нецільовий стан «більш багатообіцяючим» у порівнянні з іншим, називаються стратегіями **інформативного пошуку або евристичного пошуку**; вони розглядаються далі. Всі стратегії пошуку відрізняються тим, у якому порядку відбувається розгортання вузлів.

2.5.1 Пошук в ширину

Пошук в ширину – це проста стратегія, в якій спочатку розгортається кореневий вузол, потім – всі спадкоємці кореневого вузла, потім спадкоємці спадкоємців і т.д. Тобто при пошуку в ширину, перед тим як розгорнути вузли на наступному рівні, розгортаються всі вузли на даній глибині дерева пошуку.

Пошук в ширину може бути реалізовано шляхом виклику процедури Tree-Search з пустою периферією, яка представляє собою послідовну чергу(First-In-First-Out -- FIFO), що гарантує, що спочатку будуть розгорнуті вузли, що відвідуються першими. Інакше кажучи, до організації пошуку в глибину призводить виклик процедури Tree-Search(problem, FIFO-Queue()). В черзі FIFO передбачена вставка всіх знову сформованих спадкоємців в кінець черги, а це означає, що поверхневі вузли розгортаються раніше ніж глибокі. На рисунку 3.7 показано хід пошуку в простому бінарному дереві.

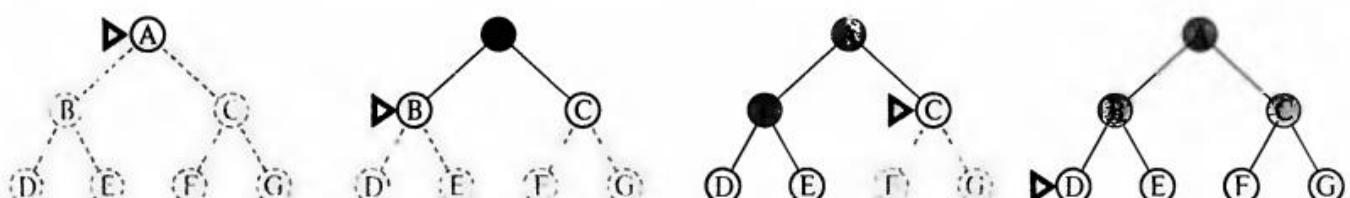


Рисунок 2.7 – Пошук в ширину в простому бінарному дереві. На кожному етапі вузол, що має бути розгорнуто наступним, позначене маркером

Проведемо оцінку пошуку в ширину з використанням чотирьох критеріїв, описаних в минулому розділі. Очевидно, що цей пошук є повним – якщо найбільш поверхневий вузол знаходиться на деякій кінцевій глибині d , то пошук в ширину в кінцевому рахунку дозволяє його знайти після розгортання більш поверхневих вузлів (за умови, що коефіцієнт розгалуження b являється кінцевим). Найбільш поверхневий цільовий вузол не обов'язково є оптимальним; формально пошук в ширину буде оптимальним, якщо ціна шляху виражається у вигляді неспадної функції глибини вузла. (Наприклад, таке припущення виправдається, якщо всі дії мають однакову ціну.)

До сих пір наданий вище опис пошуку в ширину не прогнозував ніяких неприємностей. Але така стратегія не завжди є оптимальною; щоб зрозуміти, з чим це пов'язано, треба визначити, яка кількість часу і який об'єм пам'яті потрібен для виконання пошуку. Для цього розглянемо гіпотетичний простір станів, у якому кожен стан має b спадкоємців. Корінь цього дерева породжує b вузлів на першому рівні, кожен з яких породжує ще b вузлів на другому рівні, що відповідає загальній кількості вузлів, що дорівнює b^2 . Кожен з них породжує ще b вузлів на третьому рівні, що відповідає загальній кількості вузлів, що дорівнює b^3 і т.д. А тепер припустімо, що рішення знаходиться на глибині d . В найгіршому випадку на рівні d необхідно розгорнути всі вузли, крім останнього (адже сам цільовий вузол не розгортається), що призводить до обробки $b^{d+1}-b$ вузлів на рівні $d+1$. Це означає, що загальна кількість оброблених вузлів дорівнює:

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Кожен оброблений вузол має залишатися в пам'яті, адже він або відноситься до периферії, або є предком периферійного вузла. Так, просторова складність стає такою ж, як і часова (з урахування додавання одного вузла, що відповідає кореню).

Тому досліджувачі, що проводять аналіз складності алгоритму, засмучуються(або радіють, якщо їм подобається долати складнощі), стикнувшись з експоненціальними оцінками складності, такими як $O(b^{d+1})$. В табл. 2.1 показано, з чим це пов'язано. В ній наведені вимоги до часу та до об'єму пам'яті при використанні пошуку в ширину з коефіцієнтом $b=10$ для різних значень глибини рішення d . При заповненні цієї таблиці припускалося, що в секунду може бути сформовано 10 000 вузлів, а для кожного вузла потрібно 1000 байтів пам'яті. Цим припущенням приблизно відповідають багато задач пошуку при їх рішенні на сучасному ПК (з урахуванням підвищувального або понижуючого коефіцієнта 100).

На основі табл. 3.1 можна зробити два важливих висновки. Перш за все, *при пошуку в ширину найбільш складною проблемою у порівнянні з значним часом виконання є забезпечення необхідності в пам'яті*. Затрати часу рівні 31 годині не здаються такими значними при очікуванні рішення важливої задачі з глибиною 8, але дуже мало комп'ютерів мають терабайт оперативної пам'яті, який потрібен для цього. На щастя, існують інші стратегії пошуку, які потребують менше пам'яті.

Другий висновок полягає в тому, що вимоги до часу все ще залишаються важливим фактором. Якщо задачу, що розглядається, має рішення на глибині 12, то (з урахуванням зроблених припущень) знадобиться 35 років на пошук в ширину(а насправді на будь-який неінформований пошук), щоб знайти її рішення. Узагалі, *задачі пошуку з експоненціальною складністю неможливо вирішити за допомогою неінформативних методів в усіх екземплярах задач, крім найменших*.

Таблиця 3.1. Потреби в часі та об'ємі пам'яті для пошуку в ширину.
Наведені тут дані отримані за наступних припущеннях: коефіцієнт розгалуження – $b=10$; швидкість формування вузлів – 10 000 вузлів/секунда; об'єм пам'яті 1000 байтів/вузол.

Глибина	Кількість вузлів	Час	Пам'яті
2	1100	0,11 секунди	1 мегабайт
4	111 100	11 секунд	106 мегабайт
6	10^7	19 хвилин	10 гігабайтів
8	10^9	31 година	1 терабайт
10	10^{11}	129 днів	101 терабайт
12	10^{13}	35 років	10 петабайтів
14	10^{15}	3523 роки	1 ексабайт

Пошук по критерію вартості

Пошук в ширину є оптимальним, якщо вартості всіх етапів рівні, адже в ньому завжди розгортається найбільш поверхневий нерозгорнутий вузол. З допомогою простого доповнення можна створити алгоритм, який є оптимальним при будь-якій функції визначення вартості етапу. Замість розгортання найбільш поверхневого вузла пошук **по критерію вартості** забезпечує розгортання вузла n з найменшою вартістю шляху. Зверніть увагу на те, що якщо вартості всіх етапів рівні, такий пошук ідентичний пошуку в ширину.

При пошуку по критерію вартості враховується не кількість етапів, що належать шляху, а тільки їх сумарна вартість. Тому процедура цього пошуку може увійти в нескінчений цикл, якщо виявиться, що в ній розгорнуто вузол, який має дію з нульовою вартістю, яка знову вказує на той самий стан (наприклад, дія NoOp). Можна гарантувати повноту пошуку за умови, що вартість кожного етапу більше або рівна деякої невеликої додатної константи e .

Ця умова є також достатньою для забезпечення оптимальності. Це означає, що вартість шляху завжди зростає з мірою проходження по цьому шляху. Тому перший цільовий вузол, выбраний для розгортання, являє собою оптимальне рішення. (Нагадаємо, що в процедурі Tree-Search перевірка цілі здійснюється лише до вузлів, выбраних для розгортання.) Рекомендуємо читачу спробувати використати алгоритм для пошуку найкоротшого шляху до Бухареста.

Пошук по критерію вартості направляється з урахуванням вартостей шляхів, а не значень глибини в дереві пошуку, тому його складність не може бути легко охарактеризованою в термінах b та d . Замість цього припустімо, що C^* - вартість оптимального рішення, і припустімо, що вартість кожної дії дорівнює щонайменше e . Це означає, що часова та просторова складність цього алгоритму в гіршому випадку дорівнює $O(b^{1+\lfloor C^*/e \rfloor})$, тобто може бути набагато більшою ніж b^d . Це пов'язано з тим, що процедури пошуку по критерію вартості можуть і часто виконують перевірку великих дерев, що складаються з малих етапів, перед тим як перейти до дослідження шляхів, у які входять крупні, але, можливо, більше корисні етапи. Безумовно, якщо всі вартості етапів рівні, то $b^{1+\lfloor C^*/e \rfloor}$ дорівнює b^d .

2.5.2 Пошук в глибину

При **пошуку в глибину** завжди розгортається найбільш глибокий вузол в поточній периферії дерева пошуку. Хід такого пошуку показано на рисунку 2.8. Пошук безпосередньо переходить на найбільш глибокий рівень дерева пошуку, на якому вузли не мають спадкоємців. По мірі того як ці вузли розгортаються вони видаляються з периферії, тому в подальшому пошук «оновлюється» з наступного найбільш поверхневого вузла, який все ще має недосліджених спадкоємців.

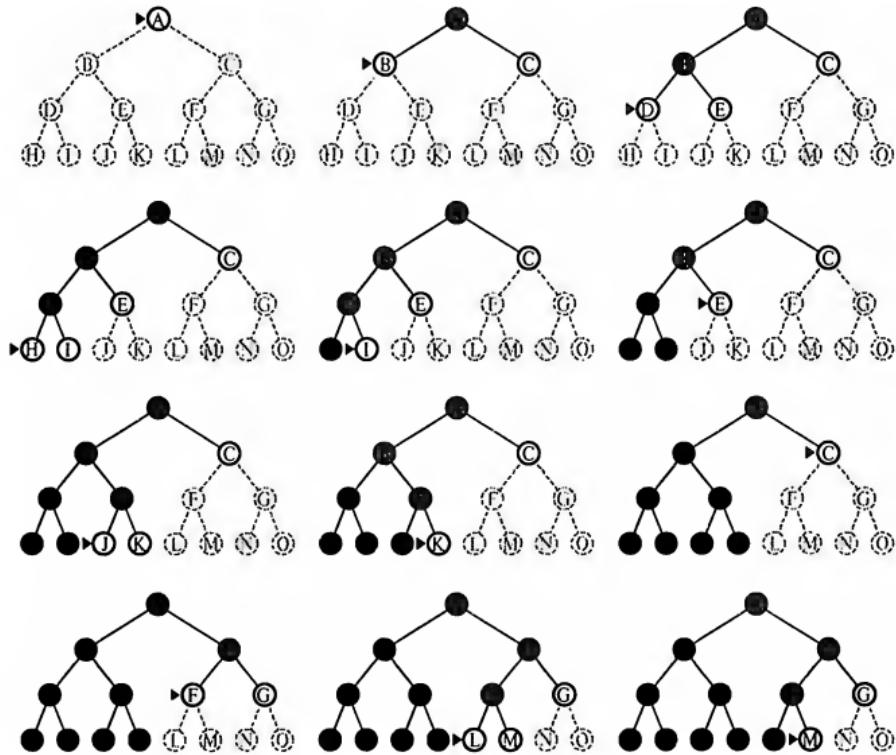


Рисунок 2.8 – Пошук в глибину в бінарному дереві. Вузли, які було розгорнуто і не маються спадкоємців в цій периферії, можуть бути видалені з пам'яті; ці вузли позначені чорним кольором. Припускається, що вузли на глибині 3 не мають спадкоємців і єдиним цільовим вузлом є М.

Ця стратегія може бути реалізована в процедурі Tree-Search за допомогою черги LIFO (Last-In-First-Out), яку ще називають *стеком*. В якості альтернативи способу реалізації на основі процедури Tree-Search пошук в глибину часто реалізують за допомогою рекурсивної функції, що викликає сама себе в кожному з дочірніх вузлів по черзі. (Рекурсивний алгоритм пошуку в глибину, в якому передбачено граничну глибину, наведено в лістингу 3.4.)

Пошук в глибину має дуже скромні вимоги до пам'яті. Він вимагає зберігання тільки єдиного шляху від кореня до листка, поряд з нерозгорнутими сестринськими вузлами, що залишилися, для кожного вузла шляху. Після того як було розгорнуто деякий вузол, він може бути видалений з пам'яті, адже скоро будуть повністю дослідженні всі його спадкоємці (див. рисунок 2.8). Для простору станів з коефіцієнтом розгалуження b і максимальною глибиною m пошук в глибину вимагає зберігання лише $bm+1$ вузлів. Використовуючи такі ж припущення, як і в табл. 3.1, і припускаючи, що вузли, що знаходяться на тій

же глибині, що і цільовий вузол, не мають спадкоємців, автори визначили, що на глибині $d=12$ для пошуку в глибину потрібно лише 118 кілобайтів замість 10 петабайтів, тобто необхідність в об'ємі зменшується приблизно в 10 мільярдів раз.

В одному з варіантів пошуку в глибину, що називають **пошуком з поверненнями**, використовується ще менше пам'яті. При пошуку з поверненнями кожен раз формується лише один спадкоємець, а не всі спадкоємці; у кожному частково розгорнутому вузлі запам'ятовується інформація про те, який спадкоємець має бути сформовано наступним. Таким чином, треба лише $O(m)$ пам'яті, а не $O(bm)$. При пошуку з поверненнями використовується ще один прийом, що дозволяє економити пам'ять(і час); ідея у тому, щоб при формування спадкоємця має безпосередньо модифікуватися опис теперішнього стану, а не викликатися його попереднє копіювання. При цьому необхідність у пам'яті зменшується до об'єму, який потрібен для зберігання тільки одного опису стану і $O(m)$ дій. Але для успішного використання цього прийому треба мати можливість відмінити кожну модифікацію при поверненні, виконаному для формування наступного спадкоємця. При вирішення задач з об'ємними описами станів, таких як роботизована збірка, використання вказаних методів модифікації станів стають найважливішим фактором успіху.

Недоліком пошуку в глибину є те, що в ньому може бути зроблено неправильний вибір і перехід в глухий кут, пов'язаний з проходженням вниз по дуже довгому(або навіть нескінченному) шляху, причому другий варіант міг би привести до рішення, що знаходиться далеко від кореня дерева. Наприклад, на рисунку 2.8 пошук в глибину вимагав би дослідження всього лівого піддерева, навіть якби цільовим вузлом був вузол С, що знаходиться в правому піддереві. А якби цільовим вузлом також був вузол J, менш прийнятний у порівнянні з вузлом С, то пошук в глибину повернув би в якості рішення саме його; це означає, що пошук в глибину не є оптимальним. Крім того, якби ліве піддерево мало необмежену глибину, але не вміщало в собі рішень, то пошук в глибину так ніколи б і не закінчився; це означає, що даний алгоритм – неповний. В

найгіршому випадку пошук в глибину формує всі $O(b^m)$ вузлів у дереві пошуку, де m – максимальна глибина будь-якого вузла. Варто зазначити, що m може бути набагато більшим у порівнянні з d (глибиною найбільш поверхневого рішення) і є нескінченим, якщо дерево має нескінченну глибину.

2.5.3 Пошук з обмеженнями в глибині

Проблему необмежених дерев можна вирішити, передбачивши використання під час пошуку в глибину заздалегідь визначеної границі глибини L . Це означає, що вузли на глибині розглядаються таким чином, ніби якби вони не мали спадкоємців. Такий підхід називається **пошуком з обмеженням в глибині**. Використання обмеження глибини дозволяє вирішити проблему нескінченого шляху. На жаль, у цього підходу також вводиться додаткове джерело неповноти, якщо буде обрано значення $L < d$, інакше кажучи, якщо найбільш поверхнева ціль виходить за границі глибини. (Така ситуація є цілком ймовірною, якщо значення d невідомо.) Крім того, пошук з обмеженнями глибину буде неоптимальним при виборі значення $L > d$. Його часова складність дорівнює $O(b^L)$, а просторова складність – $O(b^*L)$. Пошук в глибину може бути розглянуто як окремий випадок пошуку з обмеженням глибину при якому $L = \infty$.

Іноді вибір границь глибини може бути засновано на кращому розумінні задачі. Наприклад, припустимо, що на карті Румунії, що розглядається, маємо 20 міст. Тому відомо, що якщо рішення існує, то воно повинно мати довжину не більше 19; це означає, що одним з можливих варіантів є $L=19$. Але в дійсності при уважному вивчені карти можна помітити, що будь-яке місто може бути досягнуто з будь-якого іншого міста не більше ніж за 9 етапів. Це число, відоме як діаметр простору станів, дає нам кращу границю глибини, яка веде до найбільш ефективного пошуку з обмеженням глибини. Але в більшості задач прийнятна границя глибини залишається невідомою, поки не буде вирішена сама задача.

Пошук з обмеженням глибини може бути реалізовано як пристра обмеження загального алгоритму пошуку в дереві, або рекурсивного алгоритму пошуку в глибину. Псевдокод реалізації рекурсивного пошуку з

обмеженням глибини наведено в лістингу 3. Зверніть увагу на те, що пошук з обмеженнями глибини може привести до невдалих завершень двох типів: стандартне значення failure вказує на відсутність рішення, а значення cutoff говорить про те, що на даній границі рішення нема.

Лістинг 3. Рекурсивна реалізація пошуку з обмеженням глибини

Function Depth-Limited-Search (problem, limit) returns рішення result або індикатор невдачі failure|cutoff

Return Recursive-DLS(Make-Node(Initial-State[problem]),

Problem, limit)

Function Recursive-DLS(node, problem, limit) returns рішення result або індикатор невдачі failure|cutoff

cutoff_occurred? ← неправдиве значення

if Goal-Test[problem](State[node]) then return Solution(node)

else if Deptbh[node] = limit then return індикатор невдачі cutoff

else for each спадкоємець successor in Expand(node, problem) do

result ← Recursive-DLS(successor, problem, limit)

if result = cutoff then cutoff_occurred? ← правдиве значення

else if result != failure then return рішення result

if cutoff_occurred?

Then return індикатор невдачі cutoff

Else return індикатор невдачі failure

2.5.4 Пошук в глибину з ітеративним заглибленням

Пошук з ітеративним заглибленням (або, точніше, пошук в глибину з ітеративним заглибленням) являє собою загальну стратегію, що часто використовується разом з пошуком в глибину, яка дозволяє знайти кращу границю глибини. Це досягається шляхом поступового збільшення границі (яка на початку стає 0, потім 1 потім 2 і т.д.) поки не буде знайдено цільовий вузол. Це відбувається після того, як границя глибини досягає значення d , глибини найбільш поверхневого цільового вузла. Цей алгоритм наведено в лістингу 4. В

пошуку з ітеративним заглибленням поєднуються переваги пошуку в глибину і пошуку в ширину. Як і пошук в глибину, він характеризується малими вимогами у пам'яті, а саме, значенням $O(bd)$. Як і пошук в ширину, він є повним, якщо коефіцієнт розгалуження є кінцевим оптимальним, якщо вартості всіх шляхів являються собою неспадну функцію глибини вузла. На рисунку 2.9 показано чотири ітерації використання процедури Iterative-Deepening-Search до бінарного дерева пошуку, де рішення знайдено на четвертій ітерації.

Лістинг 3.5. Алгоритм пошуку з ітеративним заглибленням, в якому повторно виконується пошук з обмеженням глибини при послідовному збільшенні границі. Він завершує свою роботу після того, як знаходиться рішення, або процедура пошуку з обмеженням глибини повертає значення failure, а це означає, що рішення не існує.

function Iterative-Deepening-Search(problem) returns рішення result або індикатор невдачі failure

inputs: problem, задачі

for depth $\leftarrow 0$ to ∞ do

result \leftarrow Depth-Limited-Search(problem, depth)

if result \neq cutoff then return рішення result

Пошук з ітеративним заглибленням може на перший погляд здатися марнотратним, адже одні й ті самі вузли формуються декілька разів. Але як виявилося, такі повторні операції є не занадто «дорогими». Причина цього у тому, що в дереві пошуку з одним і тим самим (або майже тим самим) коефіцієнтом розгалуження на кожному рівні більшість вузлів знаходяться на нижньому рівні, тому не має великого значення те, що вузли на верхніх рівнях формуються багатократно. В пошуку з ітеративним заглибленням вузли на нижньому рівні (з глибиною d) формуються один раз, ті вузли, які знаходяться на попередньому формуються двічі, і т.д., поки дочірніх вузлів кореневого

вузла, які формуються d разів. Тому загальна кількість вузлів, що формується виражається формулою:

$$N(IDS) = (d)b + (d-1)b^2 + \dots + (1)b^d$$

яка відповідає часовій складності порядка $O(b^d)$. Цю кількість можна порівняти з кількістю вузлів, що формується при пошуку в ширину:

$$N(BFS) = b + b^2 + \dots + b^d + (b^{d+1} - b)$$

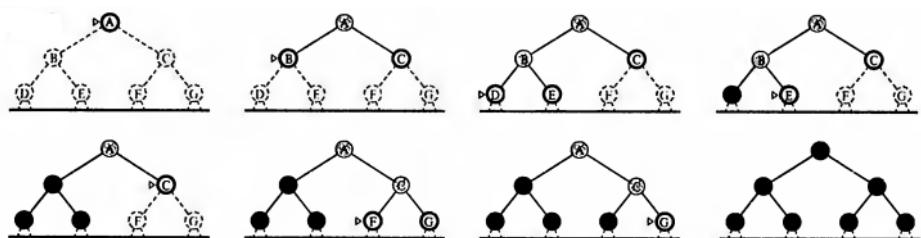
Границя 0:



Границя 1:



Границя 2:



Границя 3:

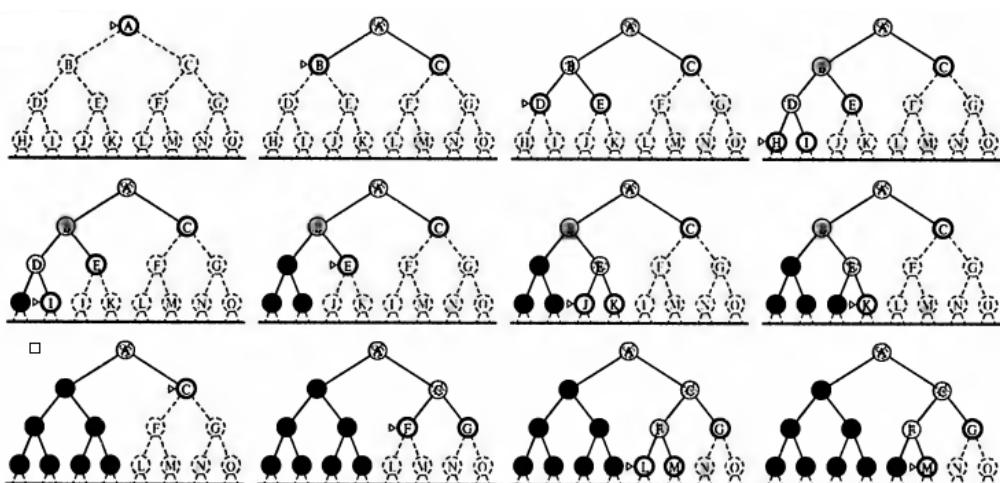


Рисунок 2.9 – Чотири ітерації пошуку з ітеративним заглибленням в бінарному дереві

Варто підмітити, що при пошуку в ширину деякі вузли формуються на глибині $d+1$, а при ітеративному заглибленні цього не відбувається. Результатом є те, що пошук з ітеративним заглибленням фактично виконується швидше, ніж пошук в ширину, незважаючи на повторне формування станів. Наприклад, якщо $b=10$, $d=5$, то відповідні оцінки кількості вузлів приймають наступні значення:

$$N(IDS) = 50 + 400 + 3000 + 20\ 000 + 100\ 000 = 123\ 450$$

$$N(BFS) = 10 + 100 + 1000 + 10\ 000 + 100\ 000 + 999\ 990 = 1\ 111\ 100$$

Взагалі, ітеративне заглиблення – це кращий метод неінформованого пошуку за тих умов, коли розглядається великий простір пошуку, а глибина рішення невідома.

Пошук з ітеративним заглибленням аналогічний пошуку в ширину в тому відношенні, що в ньому при кожній ітерації перед переходом на наступний рівень досліджується повний рівень нових вузлів. На перший погляд може здатися доцільною розробка ітеративного аналога пошуку по критерію вартості, який би успадкував би від останнього алгоритму гарантію оптимальності, дозволяючи разом з тим виключити його високі вимоги до пам'яті. Ідея у тому, щоб замість збільшення границь глибини використовувались зростаючі границі вартості шляху. Результатуючий алгоритм отримав назву пошук з ітеративним подовженням. Але, на жаль, було встановлено, що пошук з ітеративним подовженням характеризується більш важливими витратами, ніж пошук по критерію вартості.

2.5.5 Двонаправлений пошук

В основі двонаправленого пошуку лежить така ідея, що можна одночасно проводити два пошуки (в прямому напрямлені, від початкового стану, і в оберненому, від цілі), зупиняючись після того, як два процеси пошуку зустрінуться на середині (рисунок 2.10). Справа в тому, що значення $b^{d/2} + b^{d/2}$ набагато менше, ніж b^d , або, як показано на малюнку, площа двох невеликих кругів менше площині одного великого круга з центром в початку пошуку, який охоплює ціль пошуку.

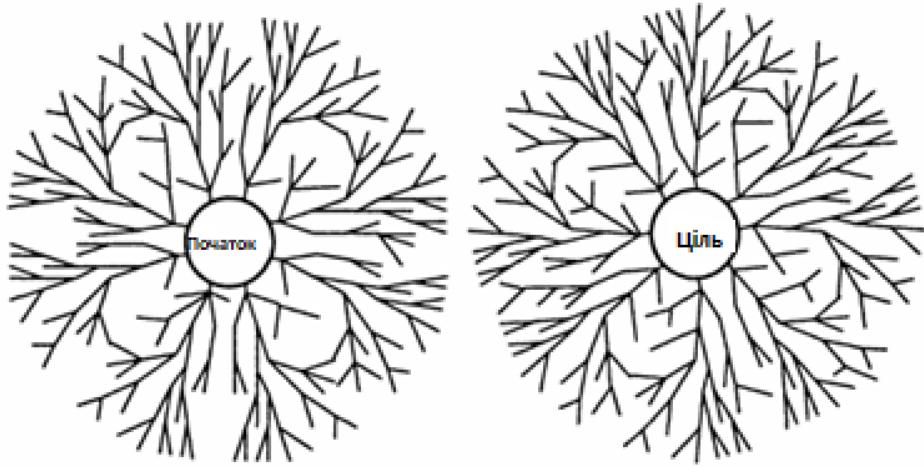


Рисунок 2.10 – Схематичне зображення двонаправленого пошуку в тому стані, коли він має скоро завершитися успішно після того як одна з гілок, що походить з початкового вузла, зустрінеться з гілкою з цільового вузла.

Двонаправлений пошук реалізовується за допомогою метода, в якому передбачено перевірка в одному чи в обох процесах пошуку кожного вузла перед його розгортанням для визначення того, чи не знаходиться він на периферії іншого дерева пошуку; у випадку позитивного результату перевірки - рішення знайдено. Наприклад якщо задача має рішення на глибині $d=6$ і в кожному напрямленні виконується пошук в ширину з послідовним розгортанням по одному вузлу, то в найгіршому випадку ці два процеси зустрінуться, якщо в кожному з них буде розгорнуто всі вузли на глибині 3, крім одного. Це означає, що при $b=10$ буде сформована загальна кількість вузлів, що дорівнює 22 000, а не 11 111 100, як при стандартному пошуку в ширину. Перевірка належності вузла до іншого дерева пошуку може бути виконана за постійний час за допомогою хеш-таблиці, тому часова складність двонаправленого пошуку визначається як $O(b^{d/2})$. В пам'яті необхідно зберігати хоча б одне з дерев пошуку, щоб можна було виконати перевірку належності до іншого дерева, тому просторова складність також визначається як $O(b^{d/2})$. Такі вимоги до простору є одним з найбільш важливих недоліків двонаправленого пошуку. Цей алгоритм є повним і оптимальним (при одинакових вартостях етапів), якщо обидва процеси пошуку виконуються в ширину; інші поєднання методів можуть характеризуватися відсутністю повноти, оптимальності або того й іншого одночасно.

Завдяки такому зменшенню часової складності двонаправлений пошук стає досить привабливим, але як організувати пошук в зворотному пошуку? Це не так легко, як здається на перший погляд. Припустімо, що батьками вузла n , що визначаються функцією $\text{Pred}(n)$, є всі ті вузли, для яких n є спадкоємцем. Для двонаправленого пошуку потрібно, щоб функція визначення батьків $\text{Pred}(n)$ була ефективною. Найпростішим є той випадок, коли всі дії в просторі станів можуть бути обернені таким чином, що $\text{Pred}(n)=\text{Succ}^{-1}(n)$, а інші випадки вимагають проявити значну винахідливість.

Розглянемо питання з тим, що мається на увазі під поняттям «ціль» при пошуку «в зворотному напрямленні від цілі». В задачах гри в вісім і пошуку маршруту в Румунії є лише один цільовий стан, тому зворотній пошук нагадує прямий пошук. Якщо ж є декілька явно перерахованих цільових станів(наприклад, два показаних на рис. 3.2 цільових стани, в яких квадрати поля не вміщають сміття), то може бути створено новий фіктивний цільовий стан. Інакше кажучи, формування деяких надлишкових вузлів можна уникнути, розглядаючи множину цільових станів як єдиний цільовий стан, кожен з батьків якого також є множиною станів, а саме, множина станів, що має відповідного спадкоємця в множині цільових станів (див. розділ 3.6).

Найбільш складним випадком для двонаправленого пошуку є така задача, в якій для перевірки цілі дано лише неявний опис деякої (можливо більшої) множини цільових станів, наприклад всіх станів, що відповідає перевірці цілі «мат» в шахах. При зворотному пошуку знадобилось би створити компактні описи всіх станів, які дозволяють поставити мат за допомогою ходу m_1 тощо; і всі ці описи треба було б звіряти з станами, що формуються при прямому пошуку. Загального ефективного способу рішення такої проблеми не існує.

Порівняння стратегій неінформованого пошуку

В табл. 2.2 наведено порівняння стратегій пошуку в термінах чотирьох критеріїв оцінки, сформованих в розділі 2.

2.6 Запобігання формування станів, що повторюються

Аж до цього моменту ми розглядали всі аспекти пошуку, але ігнорували одне з найбільш важливих ускладнень в процесі пошуку – ймовірність появи непродуктивних витрат часу при розгортанні станів, які вже зустрічалися і були розгорнуті перед цим. При вирішенні деяких завдань така ситуація ніколи не виникає; в них простір станів являє собою дерево і тому є тільки один шлях до кожного стану. Зокрема, ефективним є формулювання завдання з вісімома ферзями (в якому кожен новий ферзь поміщається на лівий порожній вертикальний ряд), його ефективність в значній мірі обумовлена саме цим – кожен стан може бути досягнуто тільки по одному шляху. А якби завдання з вісімома ферзями була сформульована таким чином, що будь-якого ферзя дозволялося б ставити на будь-яку вертикаль, то кожного стану з n ферзями можна було б досягти за допомогою $n!$ різних шляхів.

Таблиця 3.2. Оцінка стратегій пошуку, де b — коефіцієнт розгалуження; d — глибина найбільш поверхневого рішення; m — максимальна глибина дерева пошуку; l — межа глибини. Застереження, позначені малими буквами, означають наступне: ^a-повний, якщо коефіцієнт розгалуження b кінцевий; ^b — повний, якщо вартість кожного етапу $\geq \mathcal{E}$ при деякому додатному значенні E ; ^c — оптимальний, якщо вартості всіх етапів є однаковими; ^g — можна застосовувати, якщо в обох напрямках здійснюється пошук в ширину.

Характеристика	Пошук в ширину	Пошук за критерієм вартості	Пошук в глибину	Пошук з обмеженням глибини	Пошук з ітеративним заглибленням	Двохнаправлений пошук (якщо його можна застосовувати)
Повнота	Так ^a	Так ^{a,b}	Hi	Hi	Так ^a	Так ^{a,g}
Часова складність	$O(b^{d+1})$	$O(b^{l+\lceil c^*/\mathcal{E} \rceil})$ ([округлення вниз)	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Простороваскладність	$O(b^{d+1})$	$O(b^{l+\lceil c^*/\mathcal{E} \rceil})$ ([округлення вниз)	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Оптимальність	Так ^b	Так	Hi	Hi	Так ^b	Так ^{b,g}

У деяких задачах стани, що повторюються, є неминучими. До них відносяться всі задачі, в яких дії є зворотними, такі як задача пошуку маршруту і гри зі ковзаючими фішками. Дерева пошуку для цих проблем нескінчені, але якщо ми відітнемо деякі з повторюваних станів, то зможемо зменшити дерево пошуку до кінцевого розміру, формуючи тільки ту частину дерева, яка охоплює весь граф простору станів. Розглядаючи тільки частину дерева пошуку аж до

деякої постійної глибини, можна легко виявити ситуації, в яких видалення повторюваних станів дозволяє досягти експоненціального зменшення вартості пошуку. В крайньому випадку простір станів розміром $d + 1$ (рисунок 2.11, а) стає деревом з 2^d листям (рисунок 2.11, б). Більш реалістичним прикладом можуть служити прямокутні решітки, як показано на рисунок 2.11, в. В решітці кожен стан має чотирьох наступників, тому дерево пошуку, що включає повторювані стани, має 4^d листків, але існує приблизно тільки $2d^2$ різних станів з d етапами досягнення будь-якого конкретного стану. Для $d = 20$ це означає, що існує близько трильйона вузлів, але лише приблизно 800 різних станів.

Таким чином, нездатність алгоритму виявляти повторювані стани може послужити причиною того, що розв'язувана задача стане нерозв'язуваною. Така ситуація зазвичай зводиться до того, що вузол, що підлягає розгортанню, порівнюється з тими вузлами, які вже були розгорнуті; якщо виявлено збіг, то алгоритм розпізнав наявність двох шляхів в один і той же стан і може відкинути один з них.

При пошуку в глибину в пам'яті зберігаються тільки ті вузли, які лежать на шляху від кореня до поточного вузла. Порівняння цих вузлів з поточним вузлом дозволяє алгоритму виявiti циклічні шляхи, які можуть бути негайно відкинуті. Це дозволяє забезпечити, щоб кінцеві простири станів не перетворювалися в нескінченні дерева пошуку через цикли, але, на жаль, не дає можливості запобігти експоненціальному розростанню нециклічних шляхів в задачах, подібних наведеним на рисунку 2.11. Єдиний спосіб запобігання цьому полягає в тому, що в пам'яті потрібно зберігати більше вузлів. У цьому полягає фундаментальний компроміс між простором і часом. *Алгоритми, які забувають свою історію, приречені на те, щоб її повторювати.*

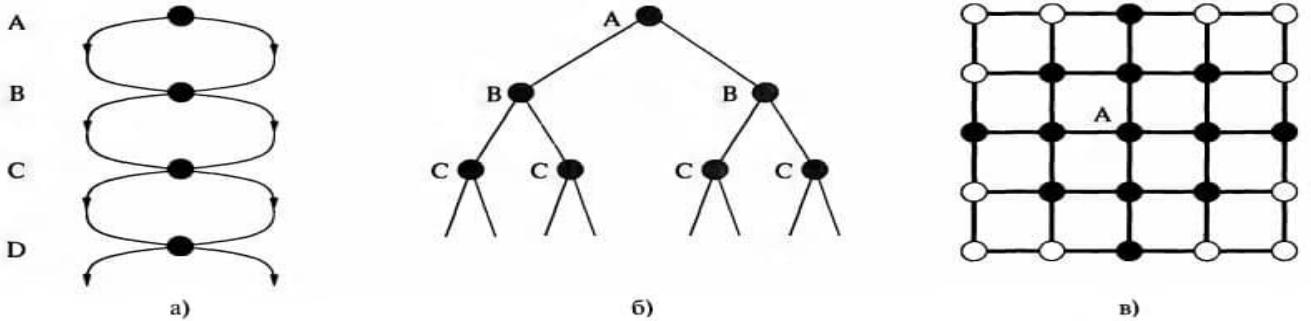


Рисунок 2.11. Простори станів, які формують експоненційно більш громіздкі дерева пошуку: простір станів, в якому є дві можливих дії, що ведуть від А до В, два - від В до С і т.д.; це простір станів містить $d + 1$ станів, де d - максимальна глибина (а); дерево пошуку, яке має 2^d гілок, відповідних 2^d шляхам через цей простір (б); Простір станів у вигляді прямокутної решітки (в); стану, що знаходиться в межах 2 етапів від початкового стану (А), позначені сірим кольором

Якщо деякий алгоритм запам'ятовує кожен стан, який він відвідав, то може розглядатися як безпосередньо граф, що досліджує, простір станів. Зокрема, можна модифікувати загальний алгоритм *Tree-Search*, щоб включити в нього структуру даних, яка називається **закритим списком**, в якому зберігається кожен розгорнутий вузол. (Периферію, що складається з нерозгорнутих вузлів, іноді називають **відкритим списком**.) Якщо поточний вузол збігається з будь-яким вузлом із закритого списку, то він не розгортається, а відкидається. Цьому новому алгоритму присвоєно назву алгоритму пошуку в графі, *Graph-Search* (лістинг 6). При вирішенні задач з багатьма періодичними станами алгоритм *Graph-Search* є набагато більш ефективним в порівнянні з *Tree-Search*. У найгіршому випадку вимоги до часу і простору пропорційні розміру простору станів. Ця величина може виявитися набагато меншою, ніж $O(b^d)$.

Питання в тому, чи оптимальний пошук в графі, залишається складним. Вище було зазначено, що поява повторюваного стану відповідає виявленню алгориттом двох шляхів в один і той же стан. Алгоритм *Graph-Search*, наведений у лістингу 6, завжди відкидає знову виявлений шлях і залишає початковий; очевидно, що якщо цей знову виявлений шлях коротше, ніж первинний, то алгоритм *Graph-Search* може втратити оптимальне рішення. На щастя, можна показати, що цього не може статися, якщо використовується або

пошук за критерієм вартості, або пошук в ширину з постійними цінами етапів; таким чином, ці дві оптимальні стратегії пошуку в дереві є також оптимальними стратегіями пошуку в графі. При пошуку з ітеративним заглибленням, з іншого боку, використовується розгортання в глибину, тому цей алгоритм цілком може пройти до деякого вузла по неоптимальному шляху, перш ніж знайти оптимальний. Це означає, що при пошуку в графі з ітеративним заглибленням необхідно перевіряти, чи не є знову виявлений шлях до вузла кращим, ніж первинний, і в разі позитивної відповіді в ньому може знадобитися переглядати значення глибини і вартості шляхів для нащадків цього вузла.

Лістинг 6. Загальний алгоритм пошуку в графі. Множина `closed` може бути реалізована за допомогою хеш-таблиці для забезпечення ефективної перевірки повторюваних станів. У цьому алгоритмі передбачається, що перший знайдений шлях до стану `s` є найменш дорогим (див. текст)

Function Graph-Search (problem, fringe) returns рішення

або індикатор невдачі failure

closed \leftarrow порожня множена

fringe \leftarrow Insert (Make – Node (Initial – State [problem]), fringe)

loop do

If Empty? (fringe) then return індикатор невдачі failure

node \leftarrow Remove – First (fringe)

If Goal – Test [problem] (State [node]) then return

Solution (node)

If State [node] не перебуває у безлічі closed then

додати State [node] до безлічі closed

Fringe \leftarrow Insert – All (Expand (node, problem), fringe)

Між іншим, використання закритого списку `closed` означає, що пошук в глибину і пошук з ітеративним поглибленням більше не мають лінійних вимог до простору. Оскільки в алгоритмі Graph-Search кожен вузол зберігається в пам'яті, деякі методи пошуку стають неможливими через недостатній обсяг пам'яті.

2.7 Евристичні алгоритми, Інформативний пошук

Евристика - галузь знання, наукова область, що вивчає специфіку творчої діяльності.

Під евристикою розуміють сукупність прийомів і методів, що полегшують і спрощують рішення пізнавальних, конструктивних, практичних задач.

Евристичний алгоритм - алгоритм рішення задачі, що включає практичний метод, який не є гарантовано точним або оптимальним, але достатній для вирішення поставленої задачі. Дозволяє прискорити вирішення задач в тих випадках, коли точне рішення не може бути знайдено.

Евристичний алгоритм - це алгоритм рішення задачі, правильність якого для всіх можливих випадків не доведена, але про який відомо, що він дає досить хороше рішення в більшості випадків. Насправді може бути навіть відомо (тобто доведено) те, що евристичний алгоритм формально невірний. Його все одно можна застосовувати, якщо при цьому він дає невірний результат тільки в окремих, досить рідкісних і добре відокремлених випадках або ж дає неточний, але все ж прийнятний результат.

Простіше кажучи, евристика - це не повністю математично обґрунтovаний (або навіть «не зовсім коректний»), але при цьому практично корисний алгоритм.

Важливо розуміти, що евристика, на відміну від коректного алгоритму розв'язання задачі, має наступні особливості:

- вона не гарантує знаходження кращого розв'язку;
- вона не гарантує знаходження розв'язку, навіть якщо він явно існує (можливий «пропуск цілі»);
- вона може дати невірний розв'язок в деяких випадках.

Загальний розглянутий нами підхід називається пошуком по першому найкращому співпадінню.

Пошук по першому найкращому співпадінню є різновидом загального алгоритму Tree-Search або Graph-Search, в якому вузол для розгортання вибирається на основі функції оцінки, $f(n)$. За традицією для розгортання вибирається вузол з найменшою оцінкою, оскільки така оцінка вимірює відстань до цілі.

Назва "пошук по першому найкращому співпадінню" узаконено традицією, але не є точною. Зрештою, якби ми дійсно могли розгортати найкращий вузол першим, то не було б і пошуку як такого; рішення задачі представляло б собою прямий хід до цілі. Єдине, що ми можемо зробити, - це вибрати вузол, який видається найкращим відповідно до функції оцінки. Якщо функція оцінки дійсно є точною, то обраний вузол справді виявиться найкращим вузлом, але фактично функція оцінки іноді виявляється малопридатною і здатною завести пошук в глухий кут. Проте, будемо дотримуватися назви "пошук по першому найкращому співпадінню", оскільки більш відповідна назва "пошук по першому співпадінню, яке можна вважати найкращим" була б досить громіздкою.

Існує ціле сімейство алгоритмів пошуку по першому найкращому співпадінню, з різними функціями оцінки. Ключовим компонентом цих алгоритмів є евристична функція, що позначається як $h(n)$:

$h(n)$ = оцінка вартості найменш дорогошого шляху від вузла n до цільового вузла.

Наприклад, у розглянутій далі задачі пошуку маршруту будемо оцінювати вартість найменш дорогошого шляху від Арада до Бухареста з допомогою відстаней по прямій до Бухареста, вимірюваних в вузлових точках маршруту від Арада до Бухареста.

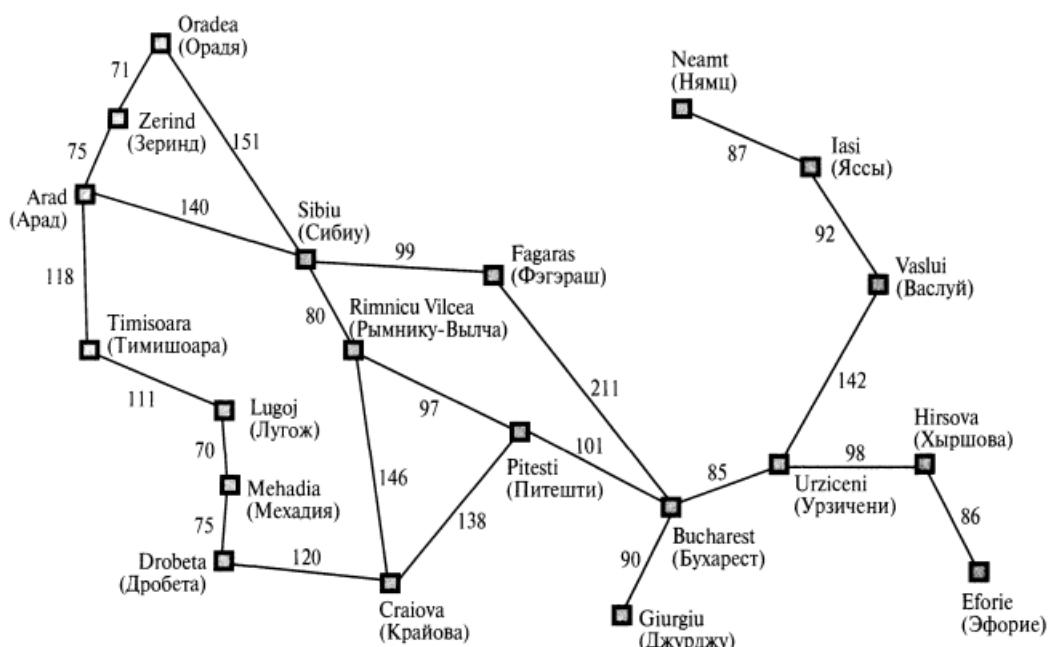
Евристичні функції (або просто евристики) являють собою найбільш загальну форму, в якій до алгоритму пошуку підключаються додаткові знання про задачу. Ми будемо визначати їх як довільні функції, пов'язані з конкретною проблемою, з одним обмеженням: якщо n - цільової вузол, то $h(n) = 0$. Далі

розглянемо два способи використання евристичної інформації для управління пошуком.

2.7.1 Жадібний пошук по першому найкращому співпадінню

При жадібному пошуку по першому найкращому співпадінню робляться спроби розгортання вузла, який розглядається як найближчий до мети на тій підставі, що він з усією ймовірністю повинен швидко привести до розв'язку. Таким чином, при цьому пошуку оцінка вузлів проводиться з використанням тільки евристичної функції: $f(n) = h(n)$.

Тепер розглянемо, як використовується цей алгоритм при розв'язанні задачі пошуку маршруту в Румунії на основі евристичної функції визначення відстані по прямій, для якої прийнято позначення h_{SLD} .



Якщо ціллю є Бухарест, то необхідно знати відстані по прямій від кожного іншого міста до Бухареста, які наведені в таблиці.

Обозначение узла	Название города	Расстояние по прямой до Бухареста	Обозначение узла	Название города	Расстояние по прямой до Бухареста
<i>Arad</i>	Арад	366	<i>Mehadia</i>	Мехадия	241
<i>Bucharest</i>	Бухарест	0	<i>Neamt</i>	Нямц	234
<i>Craiova</i>	Крайова	160	<i>Oradea</i>	Орадя	380
<i>Drobeta</i>	Дробета	242	<i>Pitesti</i>	Питешти	100
<i>Eforie</i>	Эфорие	161	<i>Rimnicu Vilcea</i>	Рымнику-Вылча	193
<i>Fagaras</i>	Фэгэраш	176	<i>Sibiu</i>	Сибиу	253
<i>Giurgiu</i>	Джурджу	77	<i>Timisoara</i>	Тимишоара	329
<i>Hirsova</i>	Хыршова	151	<i>Urziceni</i>	Урзичени	80
<i>Iasi</i>	Яссы	226	<i>Vaslui</i>	Васлуй	199
<i>Lugoj</i>	Лугож	244	<i>Zerind</i>	Зеринд	374

Наприклад, $h_{SLD}(\text{In (Arad)}) = 366$. Зверніть увагу на те, що значення h_{SLD} не можуть бути обчислені на підставі опису самої задачі. Крім того, для використання цієї евристичної функції потрібен певний досвід, що дозволяє дізнатися, яким чином значення h_{SLD} пов'язані з дійсними дорожніми відстанями, а це означає, що дана функція знаходиться практично.

На рисунку 2.12 показаний процес застосування жадібного пошуку по першому найкращому співпадінню з використанням значень h_{SLD} для визначення шляху від Араду до Бухаресту.

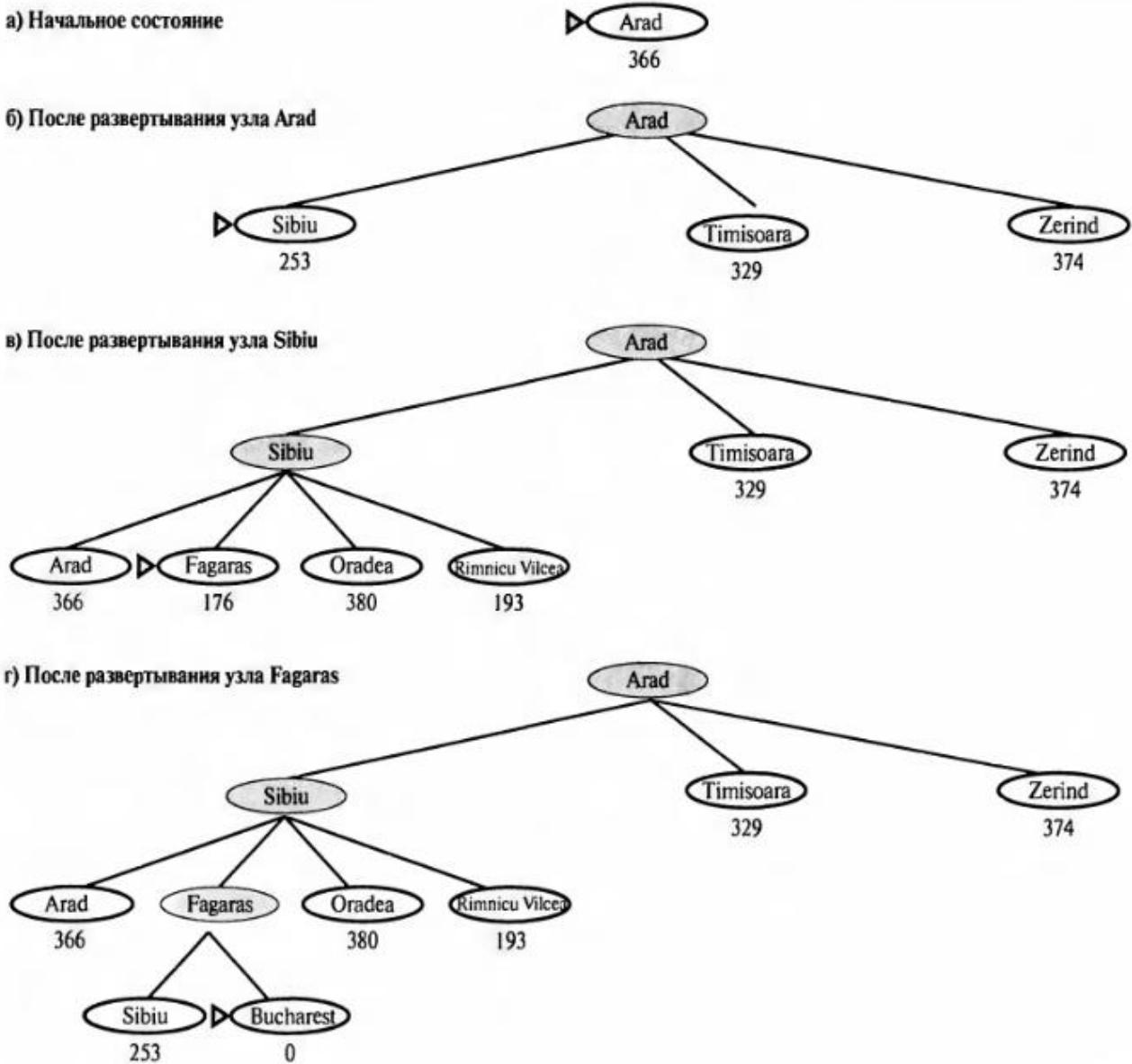


Рисунок 2.12

Першим вузлом, що підлягає розгортанню з вузла Arad, є вузол Sibiu, оскільки місто Сібіу знаходиться близче до Бухаресту, ніж міста Зерінд або Тімішоара. Наступним вузлом, що підлягає розгортанню, є вузол Fagaras оскільки тепер найближчим до Бухаресту є місто Фагараш. Вузол Fagaras в свою чергу, забезпечує формування вузла Bucharest, який є цільовим. Застосування в процесі вирішення даного конкретного завдання алгоритму жадібного пошуку по першому найкращому співпадінню з використанням функції дозволяє знайти рішення без розгортання будь-якого вузла, що не перебуває на шляху розв'язання; це означає, що вартість такого пошуку є мінімальною. Але саме знайдене рішення не оптимальне: шлях до Бухареста через міста Сібіу і Фагараш на 32 кілометри довший, ніж шлях через міста

Римніку-Вілча і Пітешті. Це зауваження показує, чому даний алгоритм називається "жадібним": на кожному етапі він намагається підійти до цілі якомога ближче (фігулярно висловлюючись, "захопити якомога більше").

Процедура мінімізації $h(n)$ вразлива до фальстарту (при її використанні іноді доводиться скасовувати початкові етапи). Розглянемо задачу пошуку шляху від міста Ясси до міста Фагараш. Ця евристична функція підказує, що в першу чергу повинен бути розгорнутий вузол міста Нямц, *Neamt*, оскільки він є найближчим до вузла *Fagaras*, але цей шлях веде у глухий кут. Рішення полягає в тому, щоб відправитися спочатку в місто Васлуй (а цей етап, відповідно до даної евристичної функції, фактично веде далі від цілі), а потім продовжувати рух через Урзічені, Бухарест і, нарешті, в Фагараш. Тому в даному випадку застосування зазначеної евристичної функції викликає розгортання НЕ-потрібних вузлів. Більш того, якщо не буде передбачено виявлення повторюваних станів, то розв'язок так ніколи не буде знайдено - процедура пошуку зациклиться між вузлами *Neamt* і *Iasi*.

Жадібний пошук по першому найкращому співпадінню нагадує пошук в глибину в тому відношенні, що цей алгоритм має бажання на шляху до мети постійно слідувати по єдиному шляху, але повертається до попередніх вузлів після потрапляння в глухий кут. Даний алгоритм страждає від тих же недоліків, що і алгоритм пошуку в глибину: він не є оптимальним, до того ж він - не повний (оскільки здатний відправитися по нескінченному шляху, так і не повернувшись, щоб випробувати інші можливості). При цьому в найгіршому випадку оцінки часової і просторової складності становлять $O(n^m)$, де m - максимальна глибина простору пошуку. Однак гарна евристична функція дозволяє істотно скоротити таку складність. Величина цього скорочення залежить від конкретної задачі і від якості евристичної функції.

2.7.2 Пошук А * мінімізація сумарної оцінки вартості рішення

Найбільш широко відомий різновид пошуку по першому найкращому співпадінню називається пошуком А * (читається як "А зірочка"). У ньому

застосовується оцінка вузлів, що об'єднує в собі $g(n)$, вартість досягнення даного вузла, та $h(n)$, вартість проходження від даного вузла до цільового:

$$f(n) = g(n) + h(n)$$

Оскільки функція $g(n)$ дозволяє визначити вартість шляхом від початкового вузла до вузла n , а функція $h(n)$ визначає оцінку вартості найменш дорогого шляху від вузла n до цільового, то справедлива наступна формула:

$f(n)$ = оцінка вартості найменш дорогого шляхи вирішення, що проходить через вузол n

Таким чином, при здійсненні спроби знайти найменш дороге рішення, мабуть, найрозумніше спочатку спробувати перевірити вузол з найменшим значенням $g(n) + h(n)$. Як виявилося, дана стратегія є більше ніж просто розумною: якщо евристична функція $h(n)$ задовольняє деяким умовам, то пошук A^* стає і повним, і оптимальним.

Опис алгоритму

A^* покроково переглядає всі шляхи, що ведуть від початкової вершини в кінцеву, поки не знайде мінімальний. Як і всі інформативні алгоритми пошуку, він переглядає спочатку ті маршрути, які «здається» ведуть до цілі. Від жодного алгоритму, який теж є алгоритмом пошуку по першому найкращому співпадінню, його відрізняє те, що при виборі вершини він враховує, крім іншого, весь пройдений до неї шлях. Складова $g(n)$ - це вартість шляху від початкової вершини, а не від попередньої, як в жадібному алгоритмі.

На початку роботи розглядаються вузли, суміжні з початковим; вибирається той з них, який має мінімальне значення $f(n)$, після чого цей вузол розкривається. На кожному етапі алгоритм оперує з множиною шляхів з початкової точки до всіх ще не розкритих (листових) вершин графа - множиною часткових рішень, - яке розміщується в черзі з пріоритетом.

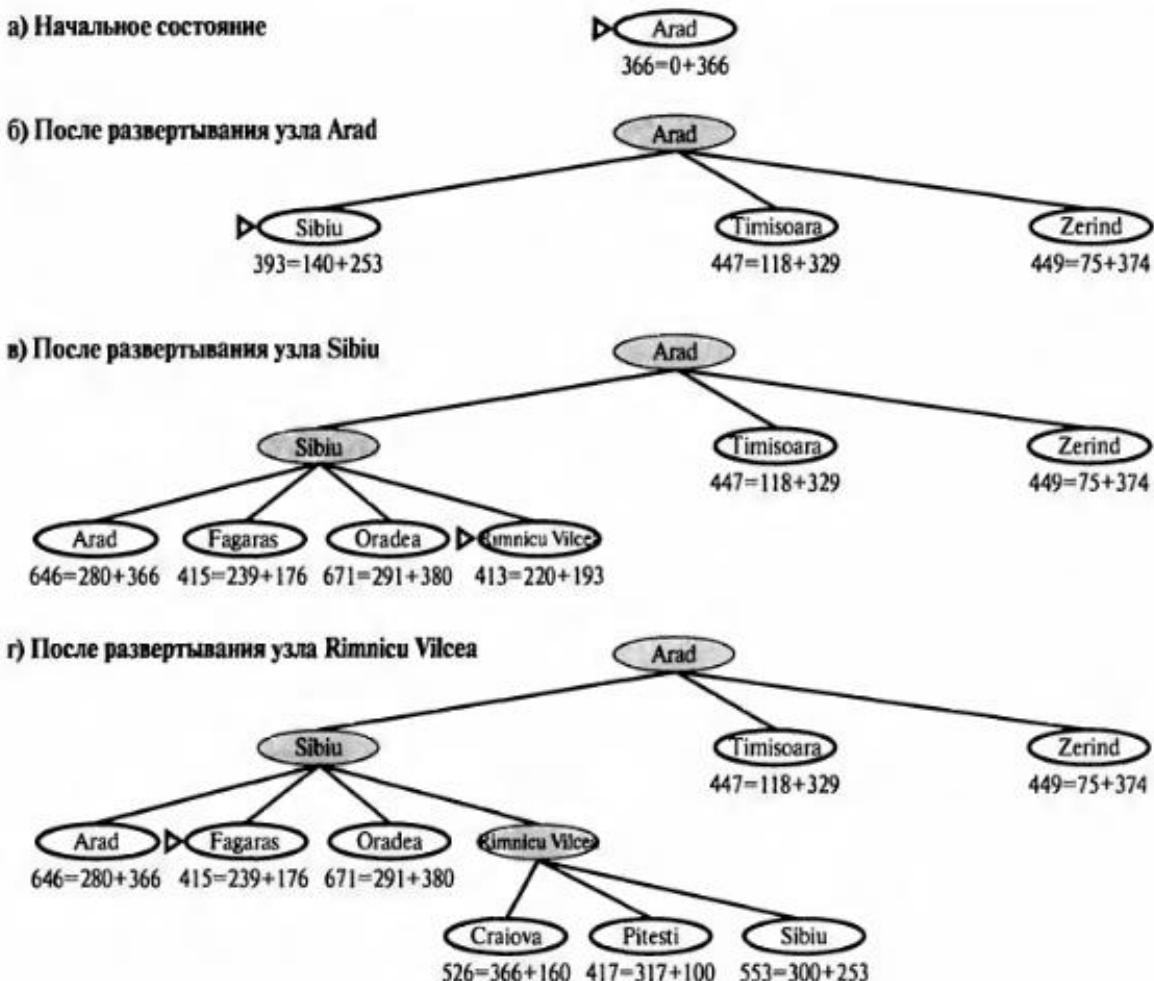
Пріоритет шляху визначається за значенням $f(n) = g(n) + h(n)$.

Алгоритм продовжує свою роботу до тих пір, поки значення $f(n)$ цільової вершини не виявиться меншим, ніж будь-яке значення в черзі, або поки все

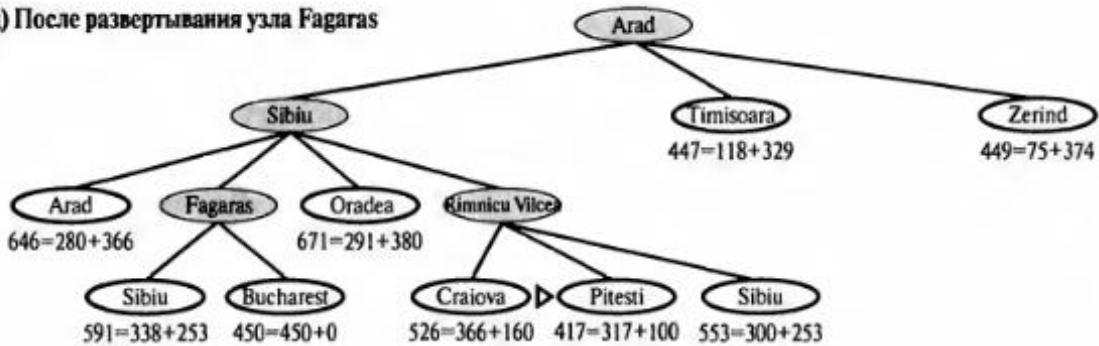
дерево не буде переглянуто. З множини рішень вибирається рішення з найменшою вартістю.

Чим менше евристика $h(n)$, тим більше пріоритет, тому для реалізації черги можна використовувати сортуючі дерева.

Розглянемо минулий приклад, з використанням алгоритму А * (рисунок 2.13).



д) Після розвертывання узла Fagaras



е) Після розвертывання узла Pitesi

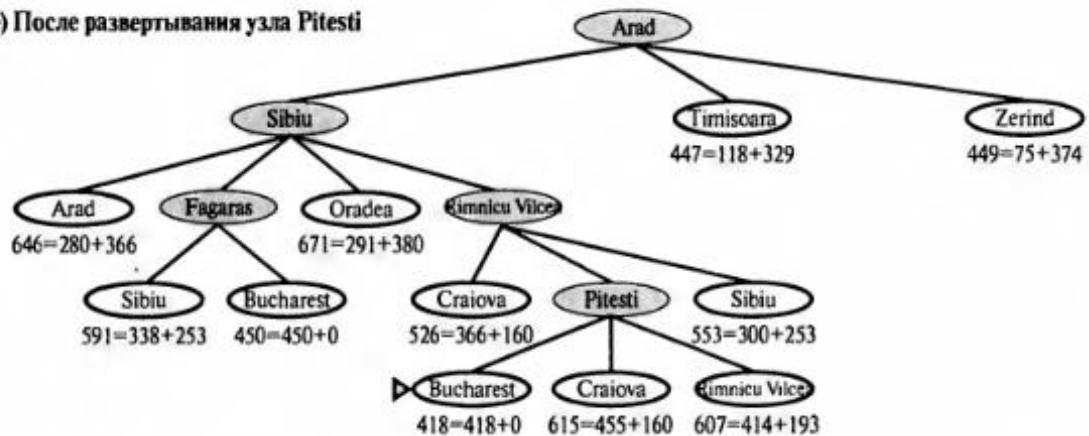


Рисунок 2.13

Множина переглянутих вершин зберігається в «closed», а ті які потребують розгляду шляху - в черзі з пріоритетом «open». Пріоритет шляху обчислюється за допомогою функції $f(x)$ всередині реалізації черги з пріоритетом.

2.8 Евристичний пошук з обмеженням обсягу пам'яті

Найпростіший спосіб скорочення потреб в пам'яті для пошуку A* полягає в застосуванні ідеї ітеративного поглиблення в контексті евристичного пошуку. Реалізація цієї ідеї призвела до створення алгоритму A* з ітеративним поглибленням (Iterative-Deepening A* - IDA*). Основна відмінність між алгоритмом IDA* і стандартним алгоритмом ітеративного поглиблення полягає в тому, що умовою зупинки розгортання є f-вартість ($g + h$), а не глибина; на кожній ітерації цим стоп-значенням є мінімальна f-вартість будь-якого вузла, що перевищує зупинне значення, досягнуте в попередній ітерації. Алгоритм IDA* є практично застосовним для вирішення багатьох задач з одиничними варгостями етапів і дозволяє уникнути істотних витрат, пов'язаних з підтримкою відсортованої черги вузлів. На жаль, цей алгоритм

характеризується такими ж складнощами, пов'язаними з використанням вартостей з дійсними значеннями, як й ітеративна версія пошуку за критерієм вартості.

2.8.1 Рекурсивний пошук за першим найкращим збігом

Рекурсивний пошук за першим найкращим збігом (Recursive Best-First Search — RBFS) - це простий рекурсивний алгоритм, в якому вживаються спроби імітувати роботу стандартного пошуку за першим найкращим збігом, але з використанням тільки лінійного простору. Цей алгоритм наведено в лістингу 7. Він має структуру, аналогічну структурі рекурсивного пошуку в глибину, але замість нескінченного проходження вниз по поточному шляху даний алгоритм контролює f -значення найкращого альтернативного шляху, доступного з будь-якого предка поточного вузла. Якщо поточний вузол перевищує цю межу, то поточний етап рекурсії скасовується і рекурсія продовжується з альтернативного шляху. Після скасування даного етапу рекурсії в алгоритмі RBFS відбувається заміна f -значення кожного вузла вздовж даного шляху найкращим f -значенням його дочірнього вузла. Завдяки цьому в алгоритмі RBFS запам'ятовується f -значення найкращого листового вузла з забутого піддерева і тому в деякий наступний момент часу може бути прийнято рішення про те, чи варто знову розгортати це піддерево. На рисунку 2.14 показано, як за допомогою алгоритму RBFS відбувається пошук шляху в Бухарест.

```

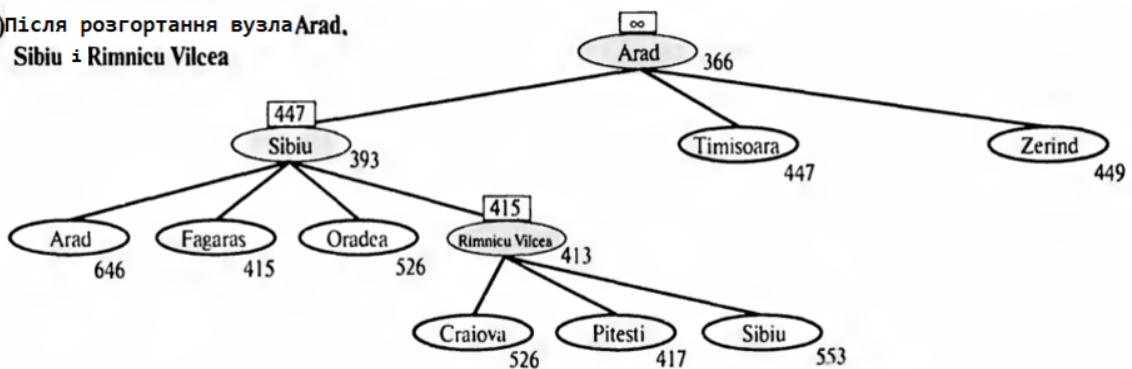
function Recursive-Best-First-Search(problem) returns рішення result
    або індикатор невдачі failure
    RBFS(problem, Make-Node(Initial-State[problem]),  $\infty$ )

function RBFS(problem, node, f_limit) returns рішення result
    або індикатор невдачі failure і нова межа f-вартості f_limit
    if Goal-Test[problem](State[node]) then return узел node
    successors  $\leftarrow$  Expand(node, problem)
    if множина вузлів спадкоємців successors пуста
        then return failure,  $\infty$ 
    for each s in successors do
        f[s]  $\leftarrow$  max(g(s) + h(s), f[node])
    repeat
        best  $\leftarrow$  вузол з найменшим f-значенням у множині successors
        if f[best] > f_limit then return failure, f[best]
        alternative  $\leftarrow$  наступне після найменшого f-значення
            у множині successors
        result, f[best]  $\leftarrow$  RBFS(problem, best,
            min(f_limit, alternative))
        if result  $\neq$  failure then return result

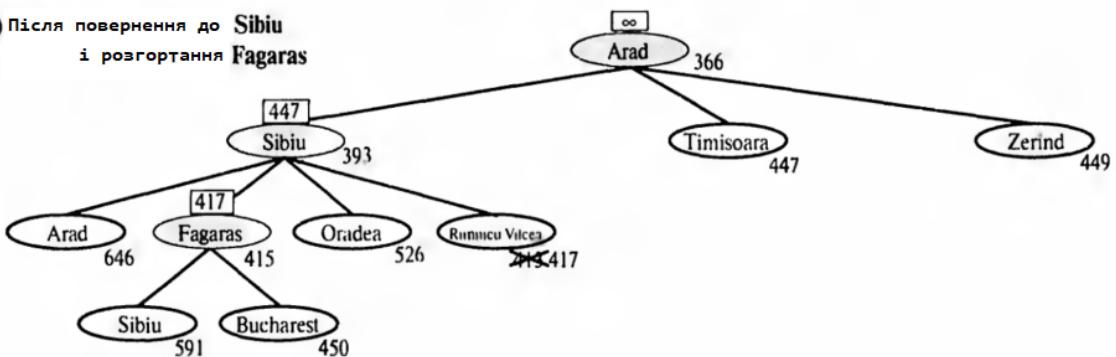
```

Лістинг 7 Алгоритм рекурсивного пошуку за першим найкращим збігом

а) Після розгортання вузла Arad,
Sibiu і Rimnicu Vilcea



б) Після повернення до Sibiu
і розгортання Fagaras



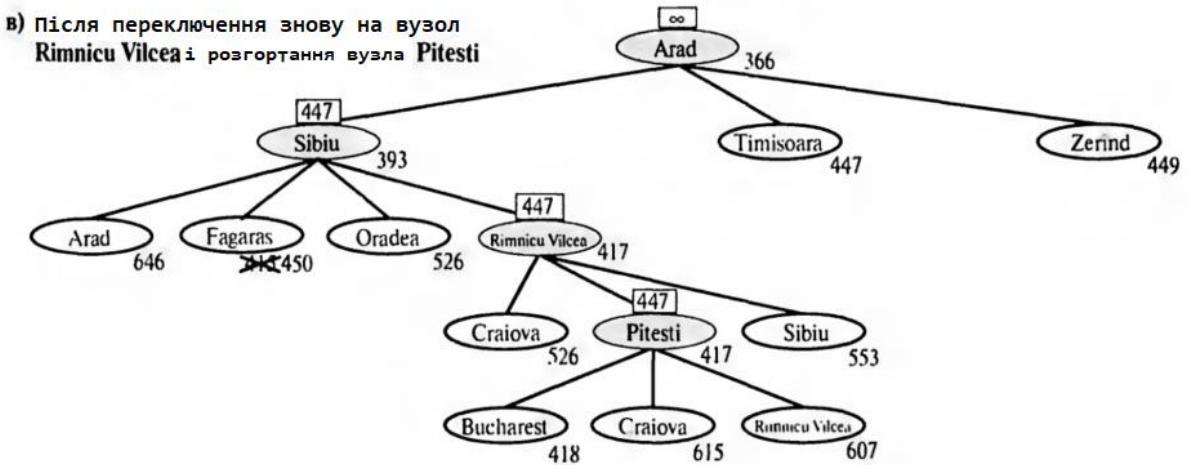


Рисунок 2.14

Етапи пошуку найкоротшого маршруту в Бухарест за допомогою алгоритму RBFS. Значення f-межі для кожного рекурсивного виклику показано над кожним поточним вузлом: шлях через вузол RimnicuVilcea, який слідує до поточного найкращого листового вузла (Pitesti,), має значення, найгірше у порівнянні з найкращим альтернативним шляхом (Fagaras,) (а); рекурсія триває і значення найкращого листового вузла забутого піддерева (417) резервується в вузлі RimnicuVilcea; потім розгортається вузол Fagaras, у результаті чого знаходиться найкраще значення листового вузла, рівне 450 (б); рекурсія триває і значення найкращого листового вузла забутого піддерева (450) резервується у вузлі Fagaras; потім розгортається вузол RimnicuVilcea; на цей раз розгортання триває в сторону Бухареста, оскільки найкращий альтернативний шлях (через вузол Timisoara,) коштує, щонайменше, 447 (в).

Алгоритм RBFS є трохи більш ефективним у порівнянні з IDA*, але все ще страждає від недоліку, пов'язаного із занадто частим повторним формуванням вузлів. У прикладі, наведеному на рисунку 2.14, алгоритм RBFS спочатку прямує шляхом через вузол RimnicuVilcea, потім "змінює рішення" і намагається пройти через вузол Fagaras, після цього знову повертається до відкинутого раніше рішення. Такі зміни рішення відбуваються в зв'язку з тим, що при кожному розгортанні поточного найкращого шляху велика ймовірність того, що його f-значення збільшиться, оскільки функція h зазвичай стає менш оптимістичною в міру того, як відбувається розгортання вузлів, більших до мети. Після того як виникає така ситуація, особливо у великих просторах

пошуку, шлях, який був наступним після найкращого, може сам стати найкращим шляхом, тому в алгоритмі пошуку доводиться виконувати повернення, щоб пройти по ньому. Кожна зміна рішення відповідає одній ітерації алгоритму IDA*, а також може вимагати багатьох повторних розгортань забутих вузлів для відтворення найкращого шляху і розгортання шляху ще на один вузол.

Як і алгоритм пошуку A*, RBFS є оптимальним алгоритмом, якщо евристична функція $h(n)$ допустима. Його просторова складність дорівнює $O(bd)$, але охарактеризувати часову складність досить важко: вона залежить і від точності евристичної функції, і від того, наскільки часто відбувалася зміна найкращого шляху в міру розгортання вузлів. І алгоритм IDA*, і алгоритм RBFS схильні до потенційного експоненціального збільшення складності, пов'язаної з пошуком в графах, оскільки ці алгоритми не дозволяють визначати наявність повторюваних станів, відмінних від тих, які знаходяться в поточному шляху. Тому дані алгоритми здатні багато разів досліджувати один і той ж стан.

Алгоритми IDA* і RBFS страждають від того недоліку, що в них використовується занадто мало пам'яті. Між ітераціями алгоритм IDA* зберігає тільки єдине число - поточну межу f-вартості. Алгоритм RBFS зберігає в пам'яті більше інформації, але кількість використованої в ньому пам'яті вимірюється лише значенням $O(bd)$: навіть якби було доступно більше пам'яті, алгоритм RBFS нездатний нею скористатися.

Тому є більш розумним використання всієї доступної пам'яті. Двома алгоритмами, які виконують цю вимогу, є пошук MA*(Memory-bounded A* - пошук A* з обмеженням пам'яті) і SMA*(Simplified MA* - спрощений пошук MA*). Алгоритм SMA* діє повністю аналогічно пошуку A*, розгортуючи найкращі листові вузли до тих пір, поки не буде вичерпана доступна пам'ять. З цього моменту він не може додати новий вузол до дерева пошуку, не знищивши старий. У алгоритмі SMA* завжди знищується найгірший листової вузол (той, який має найбільше f-значення). Як і в алгоритмі RBFS, після цього в алгоритмі SMA* значення забутого (знищеного) вузла резервується в його батьківському

вузлі. Завдяки цьому предок забутого піддерева дозволяє визначити якість найкращого шляху в цьому піддереві. Оскільки є дана інформація, в алгоритмі SMA* піддерево відновлюється, тільки якщо виявляється, що всі інші шляхи виглядають менш багатообіцяючими в порівнянні з забутим шляхом. Іншими словами, якщо всі нащадки вузла n забуті, то невідомо, яким шляхом можна слідувати від n , але все ще можна отримати уявлення про те, чи є сенс куди-небудь слідувати від n .

Як вже було сказано вище, в алгоритмі SMA* розгортається найкращий листовий вузол і видаляється найгірший листовий вузол. А що відбувається, якщо всі листові вузли мають однакове f -значення? У такому випадку може виявитися, що алгоритм вибирає для видалення і розгортання один і той самий вузол. У алгоритмі SMA* ця проблема вирішується шляхом розгортання найновішого найкращого листового вузла і видалення найстарішого найгіршого листового вузла. Ці два вузла можуть виявитися одним і тим же вузлом, тільки якщо існує лише один листовий вузол; у такому випадку дане дерево пошуку має являти собою єдиний шлях від кореня до листового вузла, що заповнює всю пам'ять. Це означає, що якщо даний листовий вузол не є цільовим вузлом, то рішення не можна досягти лише при огляді на наявний обсяг пам'яті, навіть якщо цей вузол знаходиться в оптимальному шляху вирішення. Тому такий вузол може бути відкинутий точно так, як і в тому випадку, якщо він не має наступників.

Алгоритм SMA* є повним, якщо існує якесь досяжне рішення, іншими словами, якщо d , глибина самого поверхневого цільового вузла, менше ніж обсяг пам'яті (виражений в вузлах, що зберігаються). Цей алгоритм оптимальний, якщо можна досягти будь-яке оптимальне рішення; в іншому випадку він повертає найкраще досяжне рішення. З точки зору практики алгоритм SMA* цілком може виявитися найкращим алгоритмом загального призначення для пошуку оптимальних рішень, особливо якщо простір станів є графом, вартості етапів не однакові, а операція формування вузлів є більш дорогою в порівнянні з додатковими витратами супроводу відкритих і закритих списків.

Однак при вирішенні дуже складних задач часто виникають ситуації, в яких алгоритм SMA* змушений постійно перемикатися з одного шляху вирішення на інший в межах безлічі можливих шляхів вирішення, при тому, що в пам'яті може міститися тільки невелика підмножина цієї множини. (Такі ситуації нагадують проблему пробуксовки в системах підкачки сторінок з жорсткого диска). У такому разі на повторне формування одних і тих же вузлів витрачається додатковий час, а це означає, що завдання, які були б фактично можна вирішити за допомогою пошуку A* при наявності необмеженої пам'яті, стають важковирішуваними для алгоритму SMA*. Іншими словами, через обмеження в обсязі пам'яті деякі завдання можуть ставати важковирішуваними з точки зору часу обчислення. Хоча відсутня теорія, що дозволяє знайти компроміс між витратами часу і пам'яті, створюється враження, що найчастіше уникнути виникнення цієї проблеми неможливо. Єдиним способом подолання такої ситуації стає часткова відмова від вимог до оптимальності рішення.

2.9 Навчання кращим способом пошуку

Вище було представлено кілька стратегій пошуку (пошук в ширину, жадібний пошук по першому найкращому збігу і т.д.), які були розроблені вченими і фахівцями з комп'ютерних наук. Але чи може сам агент навчатися найкращим способом пошуку? Відповідь на це питання є позитивним, а застосовуваний при цьому метод навчання спирається на важливу концепцію, яка називається *простором станів, що розглядаються на метарівні*, або метарівневим простором станів. Кожне стан в метарівневому просторі станів відображає внутрішній (обчислювальний) стан програми, яка виконує пошук в просторі станів, що розглядається на рівні об'єктів, або в **об'єктно-рівневому просторі станів**, такому як карта Румунії. Наприклад, внутрішній стан алгоритму A* включає в себе поточне дерево пошуку. Кожна дія в метарівневому просторі станів є етапом обчислення, який змінює внутрішній стан, наприклад, на кожному етапі обчислення в процесі пошуку A* розгортається один з листових вузлів, а його наступники додаються до дерева. Таким чином, рисунку 2.13, на якому показана послідовність все більших і

більших дерев пошуку, може розглядатися як шлях, що зображує в метарівневому просторі станів, де кожен стан в шляху є об'єктно-рівневим деревом пошуку.

У даний час шлях, показаний на рисунку 2.13, має п'ять етапів, включаючи один етап (розгортання вузла Fagaras), який не можна назвати надто корисним. Може виявитися, що при вирішенні більш складних завдань кількість подібних непотрібних етапів буде набагато більше, а алгоритм **метарівневого навчання** може вивчати цей досвід, щоб в подальшому уникати дослідження безперспективних піддерев. Метою навчання є мінімізація сумарної вартості вирішення задач, а також пошук компромісу між обчислювальними витратами і вартістю шляху.

2.10 Евристичні функції

Головоломка "8-puzzle" була однією з перших задач евристичного пошуку. У ході вирішення цієї головоломки потрібно пересувати фішки по горизонталі або по вертикалі на порожню ділянку до тих пір, поки отримана конфігурація не буде відповідати цільової конфігурації (рисунок 2.15).



Рисунок 2.15 – Типовий екземпляр головоломки "8-puzzle", роз'язок включає 26 етапів

Середня вартість роз'язку для сформованого випадковим чином примірника головоломки "8-puzzle" становить близько 22 етапів. Коефіцієнт розгалуження приблизно дорівнює 3. (Якщо порожній квадрат знаходиться в середині коробки, то кількість можливих ходів дорівнює чотирьом, якщо знаходиться в кутку - двом, а якщо в середині однієї зі сторін - трьом.) Це означає, що при вичерпаному пошуку на глибину 22 доводиться розглядати

приблизно $3^{22} \approx 3,1 \times 10^{10}$ станів. Відстежуючи повторювані стани, цю кількість станів можна скоротити приблизно в 170000 разів, оскільки існує тільки $9!/2 = 181440$ помітних станів, які є досяжними. Це кількість станів вже краще піддається контролю, але відповідну кількість для гри в п'ятнадцять приблизно дорівнює 10^{13} , тому для такої головоломки з більш високою складністю потрібно знайти гарну евристичну функцію. Якщо потрібно знаходити найкоротші рішення з використанням пошуку A*, то потрібна евристична функція, яка ніколи не переоцінює кількість етапів досягнення мети. Історія досліджень в області пошуку таких евристичних функцій для гри в п'ятнадцять є досить довгою, а в даному розділі розглядаються два широко використовуваних кандидати на цю роль, які описані нижче.

- $h_1 = \text{кількість фішок, що стоять не на своєму місці.}$ На рисунку 2.15 всім фішок стоять не на своєму місці, тому показаний зліва початковий стан має евристичну оцінку $h_1 = 8$. Евристична функція h_1 є допустимою, оскільки очевидно, що кожну фішку, що знаходиться не на своєму місці, необхідно перемістити щонайменше один раз.
- $h_2 = \text{сума відстаней всіх фішок від їх цільових позицій.}$ Оскільки фішки не можуть пересуватися по діагоналі, розраховується відстань, яка являє собою суму горизонтальних і вертикальних відстаней. Таку відстань іноді називають **відстань, що вимірюється в міських кварталах, або Манхетенською відстанню.** Евристична функція h_2 також є допустимої, оскільки все, що може бути зроблено в одному ході, складається лише в переміщенні однієї фішки на один етап ближче до мети. Фішки від 1 до 8 в розглянутому початковому стані відповідають такому значенню манхетенської відстані: $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$.

Як і можна було припустити, жодна з цих функцій не переоцінює справжню вартість рішення, яка дорівнює 26.

Залежність продуктивності пошуку від точності евристичної функції

Одним з критеріїв, що дозволяють охарактеризувати якість евристичної функції, є **ефективний коефіцієнт розгалуження b^* .** Якщо загальна кількість вузлів, що створюється в процесі пошуку A* при вирішенні конкретного завдання, дорівнює N, а глибина роз'язку дорівнює d, то b^* є коефіцієнтом розгалуження, який має мати однорідне дерево з глибиною d для того, щоб в ньому містилося N+1 вузлів. Тому справедлива наступна формула:

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

Наприклад, якщо алгоритм A* знаходить роз'язок на глибині 5 з використанням 52 вузлів, то ефективний коефіцієнт розгалуження дорівнює 1,92. Ефективний коефіцієнт розгалуження може змінюватися від одного примірника однієї і тієї ж задачі до іншого, але зазвичай в разі досить важких завдань залишається відносно постійним. Тому експериментальні вимірювання коефіцієнта b^* на невеликій множині завдань можуть бути гарним критерієм загальної корисності розглянутої евристичної функції. Добре спроектована евристична функція повинна мати значення b^* , близьке до 1, що дозволяє швидко вирішувати досить великі завдання.

Для перевірки евристичних функцій h_1 і h_2 було сформовано випадковим чином 1200 примірників задач з довжиною шляху роз'язку від 2 до 24 (по 100 примірників для кожного парного значення довжини) і знайшли їх роз'язок за допомогою пошуку з ітеративним поглибленнем і пошуку в дереві за алгоритмом A* з застосуванням евристичних функцій h_1 і h_2 . Дані про середню кількість вузлів, розгорнутих при використанніожної стратегії та ефективному коефіцієнту розгалуження, наведені в табл. 4.2. Ці результати показують, що евристична функція h_2 краще ніж h_1 і набагато краще в порівнянні з використанням пошуку з ітеративним поглибленнем. Стосовно знайдених рішень з довжиною 14 з застосуванням пошуку A* з евристичною функцією h_2 стає в 30 000 разів ефективнішим в порівнянні з неінформованим пошуком з ітеративним поглибленнем.

Таблиця 4.2. Порівняння значень вартості пошуку та ефективного коефіцієнта розгалуження для алгоритмів Iterative-Deepening-Search і A* з h_1 , h_2 . Дані усереднювалися по 100 екземплярів завдання 8-puzzle стосовно до різних значень довжини рішення

d	Вартість пошуку			Ефективний коефіцієнт розгалуження		
	IDS	A*(h ₁)	A*(h ₂)	IDS	A*(h ₁)	A*(h ₂)
2	10	6	6	2,45	1,79	1,79
4	112	13	12	2,87	1,48	1,45
6	680	20	18	2,73	1,34	1,30
8	6384	39	25	2,80	1,33	1,24
10	47127	93	39	2,79	1,38	1,22
12	3644035	227	73	2,78	1,42	1,24
14	—	539	113	—	1,44	1,23
16	—	1301	211	—	1,45	1,25
18	—	3056	363	—	1,46	1,26
20	—	7276	676	—	1,47	1,27
22	—	18094	1219	—	1,48	1,28
24	—	39135	1641	—	1,48	1,26

Інтерес представляє питання про те, чи завжди евристична функція h_2 краще ніж h_1 . Відповідь на це питання є позитивною. На підставі визначень цих двох евристичних функцій можна легко дійти до висновку, що для будь-якого вузла n справедливий вираз $h_2(n) \geq h_1(n)$. Таким чином, можна стверджувати, що евристика h_2 **домінує** над h_1 . Домінування пов'язане з ефективністю: при пошуку A^* з використанням функції h_2 ніколи не відбувається розгортання більшої кількості вузлів, ніж при пошуку A^* з використанням h_1 (можливо, за винятком декількох вузлів з $f(n) = C^*$). Доказ цього твердження є нескладним. Нагадаємо, що кожен вузол із значенням $f(n) < C^*$ повинен бути розгорнутий. Це аналогічно твердженню, що повинен бути напевно розгорнутий кожен вузол з значенням $h(n) < C^* - g(n)$. Але оскільки для всіх вузлів значення h_2 , хоча б не менше значення h_1 то кожен вузол, який повинен бути напевно розгорнуто в пошуку A^* з h_2 , буде також напевно розгорнуто при пошуку з h_1 а застосування евристичної функції h_1 може до того ж викликати і розгортання інших вузлів. Тому завжди краще використовувати евристичну функцію з більш високими значеннями, при тих умовах, що вона не буде переоцінювати довжину шляху вирішення і що час обчислення цієї евристичної функції не надто довгий.

2.10.1 Складання допустимих евристичних функцій

Вище було показано, що евристичні функції h_1 (в якій використовується кількість фішок, що стоять не на своєму місці) і h_2 (в якій використовується

Манхетенська відстань) є досить непоганими евристичними функціями для завдання 8-puzzle і що функція h_2 - з них найкраща. Але на підставі чого саме була запропонована функція h_2 ? Чи можливо, щоб комп'ютер міг скласти деяку евристичну функцію механічно?

Евристичні функції h_1 і h_2 є оцінками довжини шляху, що залишився, для завдання 8-puzzle, але вони, крім того, повертають ідеально точні значення довжини шляху для спрощених версій цієї гри. Якби правила 8-puzzle змінилися таким чином, щоб будь-яку фішку можна було пересувати куди завгодно, а не тільки на сусідній порожній квадрат, то евристична функція h_1 поверталася б точну кількість етапів в найкоротшому рішенні. Аналогічним чином, якщо б будь-яку фішку можна було переміщати на один квадрат в будь-якому напрямку, навіть на зайнятий квадрат, то точну кількість етапів в найкоротшому рішенні поверталася б евристична функція h_2 . Завдання з меншою кількістю обмежень на можливі дії називається **ослабленим завданням**. *Вартість оптимального рішення ослабленою завдання є допустимою евристикою для початкового завдання.* Така евристична функція є допустимою, оскільки оптимальне рішення початкової задачі, за визначенням, є також рішенням ослабленого завдання і тому має бути, щонайменше, таким же великим по вартості, як і оптимальне рішення ослабленого завдання. Оскільки значення похідної евристичної функції є точною вартістю рішення ослабленого завдання, ця функція має підкорятися нерівності трикутника і тому повинна бути **спадкоємною**.

Якщо визначення завдання записано на якісь формальній мові, то існує можливість формувати ослаблені завдання автоматично. Наприклад, якщо дії в 8-puzzle описані в такий спосіб:

Фішка може бути перенесена з квадрата A в квадрат B, якщо квадрат A є суміжним по горизонталі або по вертикалі з квадратом B і квадрат B порожній.

то можуть бути сформовані три ослаблені завдання шляхом видалення одного або обох з наведених вище умов.

- a) Фішка може бути перенесена з квадрата А в квадрат В, якщо квадрат А є суміжним з квадратом В.
- б) Фішка може бути перенесена з квадрата А в квадрат В, якщо квадрат В порожній.
- в) Фішка може бути перенесена з квадрата А в квадрат В.

На підставі ослабленою завдання а) можна вивести функцію h_2 (Манхетенську відстань). В основі цих міркувань лежить те, що h_2 повинна являти собою правильну оцінку, якщо кожна фішка пересувається до місця її призначення по черзі. На підставі ослабленою завдання в) можна отримати евристичну функцію h_1 (фішки, що стоять не на своїх місцях), оскільки ця оцінка була б правильною, якби фішки можна було пересувати в призначене для них місце за один етап. Зверніть увагу на те, що тут суттєвою є можливість вирішувати ослаблені завдання, що створюються за допомогою вказаного методу, фактично без будь-якого пошуку, оскільки ослаблені правила забезпечують декомпозицію цієї задачі на вісім незалежних підзадач. Якби було важко вирішувати і ослаблене завдання, то був би дорогим сам процес отримання значень відповідної евристичної функції. (Слід зазначити, що ідеальну евристичну функцію можна також отримати, просто дозволивши функції h "потайки" виконувати повний пошук в ширину. Таким чином, при створенні евристичних функцій завжди доводиться шукати компроміс між точністю і часом обчислення).

Для автоматичного формування евристичних функцій на основі визначень завдань з використанням методу "ослабленого завдання" і деяких інших методів може застосовуватися програма під назвою Absolver. Програма Absolver склала для завдання 8-puzzle нову евристичну функцію, кращу в порівнянні з усіма існуючими раніше евристичними функціями, а також знайшла першу корисну евристичну функцію для знаменитої головоломки "кубик Рубика".

Одна з проблем, що виникають при складанні нових евристичних функцій, полягає в тому, що часто не вдається отримати евристичну функцію, яка була б "кращою в усіх відношеннях" у порівнянні з іншими. Якщо для вирішення якого-небудь завдання може застосовуватися ціла колекція

допустимих евристичних функцій і жодна з них не домінує над будь-якої з інших функцій, то яка з цих функцій повинна бути обрана? Виявилося, що такий вибір робити не потрібно, оскільки можна взяти від них усіх найкраще, визначивши такий критерій вибору:

$$h(n) = \max\{h_1(n), \dots, h_m(n)\}$$

У цій евристиці використовується та функція, яка є найбільш точної для розглянутого вузла. Оскільки евристичні функції, що входять в склад евристики h , є допустимими, сама ця функція також є допустимою; крім того, можна легко довести, що функція h спадкоємна. До того ж h домінує над усіма наближеними функції, які входять до її складу.

Допустимі евристичні функції можуть бути також виведені на основі вартості рішення **підзадачі** даного конкретного завдання. Наприклад, на рис. 4.6 показана підзадача для екземпляра 8-puzzle, наведеного на рис. 4.5. ця підзадача стосується переміщення фішок 1, 2, 3, 4 в їх правильні позиції. Очевидно, що вартість оптимального вирішення цієї підзадачі є нижньою межею вартості рішення повного завдання. Як виявилося, така оцінка в деяких випадках є набагато більш точної в порівнянні з манхетенською відстанню.

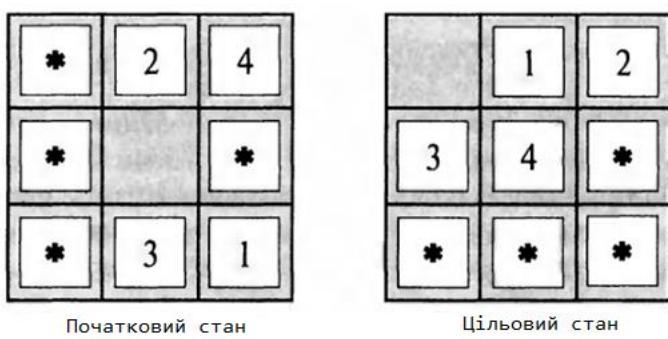


Рис. 4.6. Підзадача для екземпляра 8-puzzle, показаного на рис. 4.5.

Завдання полягає в тому, щоб пересунути фішки 1, 2, 3 і 4 в їх правильні позиції, не турбуючись про те, що станеться з іншими фішками

В основі **баз даних з шаблонами** лежить така ідея, що зазначені точні вартості рішень потрібно зберігати для кожного можливого екземпляра підзадачі - в нашему прикладі дляожної можливої конфігурації з чотирьох фішок і порожнього квадрата. (Слід зазначити, що при виконанні завдання

щодо вирішення цієї підзадачі місцезнаходження інших чотирьох фішок не потрібно брати до увагу, але ходи з цими фішками слід враховувати у вартості рішення.) Після цього обчислюється допустима евристична функція h_{DB} (DB - Data Base) для кожного повного стану, що зустрівся в процесі пошуку, шляхом вибірки даних для відповідної конфігурації підзадачі з бази даних. Сама база даних заповнюється шляхом зворотного пошуку з цільового стану і реєстрації вартості кожного нового шаблону, що зустрівся; витрати на цей пошук окуплюються за рахунок успішного вирішення в подальшому багатьох нових екземплярів задачі.

Вибір фішок 1-2-3-4 є досить довільним; можна було б також створити бази даних для фішок 5-6-7-8, 2-4-6-8 і т.д. За данимиожної бази даних формується допустима евристична функція, а ці евристичні функції можна складати в загальну евристичну функцію, як було описано вище, застосовуючи їх максимальне значення. Складова евристична функція такого виду є набагато більш точної в порівнянні з манхетенською відстанню; кількість вузлів, що виробляються при вирішенні сформованих випадковим чином примірників завдання гри в п'ятнадцять, може бути зменшено приблизно в 1000 разів.

Цікавим є питання про те, чи можна складати значення евристичних функцій, отриманих з баз даних 1-2-3-4 і 5-6-7-8, оскільки очевидно, що відповідні дві підзадачі не перекриваються. Чи буде при цьому все ще отримана допустима евристична функція? Відповідь на це питання є негативним, оскільки в рішеннях підзадачі 1-2-3-4 і підзадачі 5-6-7-8 для даного конкретного стану повинні напевно присутні деякі загальні ходи. Справа в тому, що малоймовірна ситуація, при якій фішки 1-2-3-4 можна було б пересунути на свої місця, не чіпаючи фішок 5-6-7-8, і навпаки. Але що буде, якщо не враховувати ці ходи? Іншими словами, припустимо, що реєструється не загальна вартість рішення підзадачі 1-2-3-4, а тільки кількість ходів, в яких зачіпаються фішки 1-2-3-4. В такому випадку можна легко визначити, що сума цих двох вартостей все ще є нижнєю межею вартості рішення всієї задачі. Саме ця ідея лежить в основі **баз даних з непересічними шаблонами**. Застосування таких баз даних дозволяє вирішувати випадково вибрані екземпляри завдання

гри в п'ятнадцять за кілька мілісекунд - кількість сформованих вузлів скорочується приблизно в 10 000 разів у порівнянні з використанням манхетенської відстані. Для завдання гри в 24 може бути досягнуто прискорення приблизно в мільйон разів.

Бази даних з непересічними шаблонами успішно застосовуються для рішення головоломок з легкими фішками, оскільки сама задача може бути розділена таким чином, що кожен хід впливає лише на одну підзадачу, так як одночасно відбувається переміщення тільки однієї фішки. При вирішенні таких задач, як кубик Рубика, такий поділ не може бути виконано, оскільки кожен хід впливає на стан 8 або 9 з 26 елементів кубика. В даний час немає повного розуміння того, як можна визначити бази даних з непересічними шаблонами для подібних завдань.

2.10.2 Вивчення евристичних функцій на основі досвіду

Передбачається, що евристична функція $h(n)$ оцінює вартість рішення, починаючи від стану, пов'язаного з вузлом n . Як може деякий агент скласти подібну функцію? Одне з рішень було приведено раніше, а саме: агент повинен сформулювати ослаблені завдання, для яких може бути легко знайдено оптимальне рішення. Ще один підхід полягає в тому, що агент повинен навчатися на підставі отриманого досвіду. Тут під "отриманням досвіду" мається на увазі, наприклад, рішення великої кількості примірників 8-puzzle. Кожне оптимальне рішення задачі 8-puzzle надає приклади, на основі яких можна вивчати функцію $h(n)$. Кожен приклад складається з стану, взятого з шляху вирішення, і фактичної вартості шляху від цієї точки до рішення. На основі даних прикладів за допомогою алгоритму **індуктивного навчання** може бути сформована певна функція $h(n)$, здатна (при щасливому збігу обставин) передбачати вартості рішень для інших станів, які виникають під час пошуку.

Методи індуктивного навчання діють найкраще, коли в них враховуються **характеристики стану**, релевантні для оцінки цього стану, а не просто загальний опис стану. Наприклад, характеристика "кількості фішок, що стоять не на своїх місцях" може бути корисною при прогнозі фактичного віддалення

деякого стану від мети. Наземо цю характеристику $x_1(n)$. Наприклад, можна взяти 100 сформованих випадковим чином конфігурацій головоломки 8-puzzle і зібрати статистичні дані про їх фактичних вартостей рішень. Допустимо, це дозволяє виявити, що при $x_1(n)$, що дорівнює 5, середня вартість рішення становить близько 14. Наявність таких даних дозволяє використовувати значення x_1 для передбачення значень функції $h(n)$. Безумовно, можна також застосовувати відразу кілька характеристик. Другою характеристикою, $x_2(n)$, може бути "кількість пар суміжних фішок, які є також суміжними в цільовому стані". Яким чином можна скомбінувати значення $x_1(n)$ і $x_2(n)$ для передбачення значення $h(n)$? Загальноприйнятий підхід полягає в використанні лінійної комбінації, як показано нижче.

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

Константи c_1 і c_2 коригуються для досягнення максимальної відповідності умовам фактичними даними про вартість рішень. Передбачається, що константа c_1 повинна бути позитивною, а c_2 - негативною.

2.11 Алгоритми локального пошуку і задачі оптимізації

Алгоритми пошуку, що розглядалися досі, призначалися для систематичного дослідження просторів пошуку. Така систематичність досягається завдяки тому, що один або декілька шляхів зберігається в пам'яті і проводиться реєстрація того, які альтернативи були досліджені в кожній точці впродовж цього шляху, а які ні. Після того, як ціль знайдена, шлях до цієї цілі складає шуканий розв'язок цієї задачі.

Але при розв'язанні багатьох задач шлях до цілі не представляє інтересу. Наприклад, в задачі з вісімома ферзями важлива лише остаточна конфігурація ферзів, а не порядок, в якому вони були поставлені на дошку. До цього класу задач відносять багато важливих застосувань, такі як проектування інтегральних схем, розробка плану цеху, складання виробничого розкладу, автоматичне програмування, оптимізація мережі зв'язку, складання маршруту транспортного засобу і управління портфелем акцій.

Якщо шлях до цілі не представляє інтересу, то можуть розглядатися алгоритми іншого класу, в яких взагалі не потрібні будь-які дані про шляхи. Алгоритми **локального пошуку** діють з урахуванням єдиного поточного стану (а не численних шляхів) і зазвичай передбачають тільки перехід в стан, сусідній по відношенню до поточного стану. Як правило, інформація про шляхи, пройдені в процесі такого пошуку, не зберігається.Хоча алгоритми локального пошуку не передбачають систематичне дослідження простору станів (не є систематичними), вони володіють двома важливими перевагами: по-перше, в них використовується дуже невеликий об'єм пам'яті, до того ж зазвичай постійний, і, по-друге, вони часто дозволяють знаходити прийнятні рішення у великих або нескінченних (безперервних) просторах станів, для яких систематичні алгоритми незастосовні.

Окрім пошуку цілей, алгоритми локального пошуку є корисними для розв'язку чистих **задач оптимізації**, призначення яких полягає в пошуку стану, найкращого з точки зору **цільової функції**. Багато задач оптимізації не вписуються в "стандартну" модель пошуку, представлена в главі 3. Наприклад, природа передбачила таку цільову функцію (придатність для репродукції), що дарвінівська еволюція може розглядатися як спроба її оптимізації, але в цій задачі оптимізації немає компонентів "перевірка цілі" і "вартість шляху".

Автори дійшли висновку, що для розуміння суті локального пошуку дуже корисно розглянути **ландшафт простору станів** (подібний до показаного на рис. 4.7). Цей ландшафт характеризується і "місцезнаходженням" (яке визначається станом), і "підвищенням" (яке визначається значенням евристичної функції вартості або цільової функції). Якщо підвищення відповідає вартості, то задача полягає в пошуку найглибшої долини – **глобального мінімуму**, а якщо підвищення відповідає цільової функції, то задача полягає в пошуку найвищого піку – **глобального максимуму**. (Мінімум і максимум можна поміняти місцями, узявши їх із зворотними знаками.) Алгоритми локального пошуку досліджують такий ландшафт. Алгоритм **повного** локального пошуку завжди знаходить мету, якщо вона існує, а **оптимальний** алгоритм завжди знаходить глобальний мінімум/максимум.



Рис. 4.7. Ландшафт одновимірного простору станів, в якому підвищення відповідає цільовій функції. Задача полягає в пошуку глобального максимуму. Як позначено стрілкою, в процесі пошуку за принципом "підйому до вершини" здійснюються спроби модифікації поточного стану з метою його поліпшення. Різні топографічні особливості ландшафту визначені в тексті

2.11.1 Пошук зі сходженням до вершини

Алгоритм пошуку зі **сходженням до вершини** показаний в лістингу 4.2. Він являє собою звичайний цикл, в якому постійно відбувається переміщення в напрямку зростання деякого значення, тобто підйом. Робота цього алгоритму закінчується після досягнення "піку", в якому жоден з сусідніх станів не має більш високого значення. У цьому алгоритмі не передбачено супроводу дерева пошуку, тому в структурі даних поточного вузла потрібно реєструвати тільки стан і значення цільової функції, що відповідає йому. У алгоритмі зі сходженням до вершини не здійснюється прогнозування за межами станів, які є безпосередньо сусідніми по відношенню до поточного стану. Це нагадує спробу альпініста, що страждає від амнезії, знайти вершину гори Еверест в густому тумані.

Лістинг 4.2. Алгоритм пошуку зі сходженням до вершини (версія з найбільш крутим підйомом), який є найфундаментальнішим методом локального пошуку. На кожному етапі поточний вузол замінюється найкращим сусіднім вузлом; у цій версії таким є вузол з максимальним значенням Value, але якщо використовується евристична оцінка вартості

h, то може бути передбачено пошук сусіднього вузла з мінімальним значенням **h**

function Hill-Climbing (problem) **returns** стан, який являє собою локальний максимум

inputs: problem, задача

local variables: current, вузол

neighbor, вузол

current \leftarrow Make-Node (Initial-State [problem])

loop do

neighbor \leftarrow наступник вузла current з найвищим значенням

if Value[neighbor] \leq Value[current] **then return**

State[current]

current \leftarrow neighbor

Для ілюстрації пошуку зі сходженням до вершини скористаємося задачею з **вісімома ферзями**, яка представлена на с. 118. У алгоритмах локального пошуку зазвичай застосовується **формулювання повного стану**, в якому в кожному стані на дошці є вісім ферзів, по одному ферзю в кожному стовпці. Функція визначення наступника повертає усі можливі стани, що формуються шляхом переміщення окремого ферзя в іншу клітину одного і того ж стовпця (тому кожен стан має $8 \times 7 = 56$ наступників). Евристична функція вартості **h** визначає кількість пар ферзів, які атакують один одного або пряму, або непряму (атака називається непрямою, якщо на одній горизонталі, вертикалі або діагоналі стоять більше двох ферзів). Глобальний мінімум цієї функції стає рівним нулю, і це відбувається тільки в ідеальних розв'язках. На рис. 4.8, а показаний стан зі значенням **h=17**. На цьому рисунку також показані значення усіх наступників цього стану, притому що найкращі наступники мають значення **h=12**. Алгоритми зі сходженням до вершини зазвичай передбачають

випадковий вибір у множині найкращих наступників, якщо кількість наступників більше одного.

Пошук зі сходженням до вершини іноді називають **жадібним локальним пошуком**, оскільки в процесі його виконання відбувається захоплення найліпшого сусіднього стану без попередніх міркувань про те, куди слід відправитись далі. Жадібність вважається одним з семи смертних гріхів, але, як виявилося, жадібні алгоритми часто показують дуже високу продуктивність. Під час пошуку зі сходженням до вершини часто відбувається дуже швидке просування у напрямі до розв'язку, оскільки зазвичай буває надзвичайно легко покращити поганий стан. Наприклад, зі стану, показаного на рис. 4.8, а, достатньо зробити лише п'ять ходів, щоб досягнути стану, показаного на рис. 4.8, б, який має оцінку $h=1$ і дуже близький до одного з розв'язків.

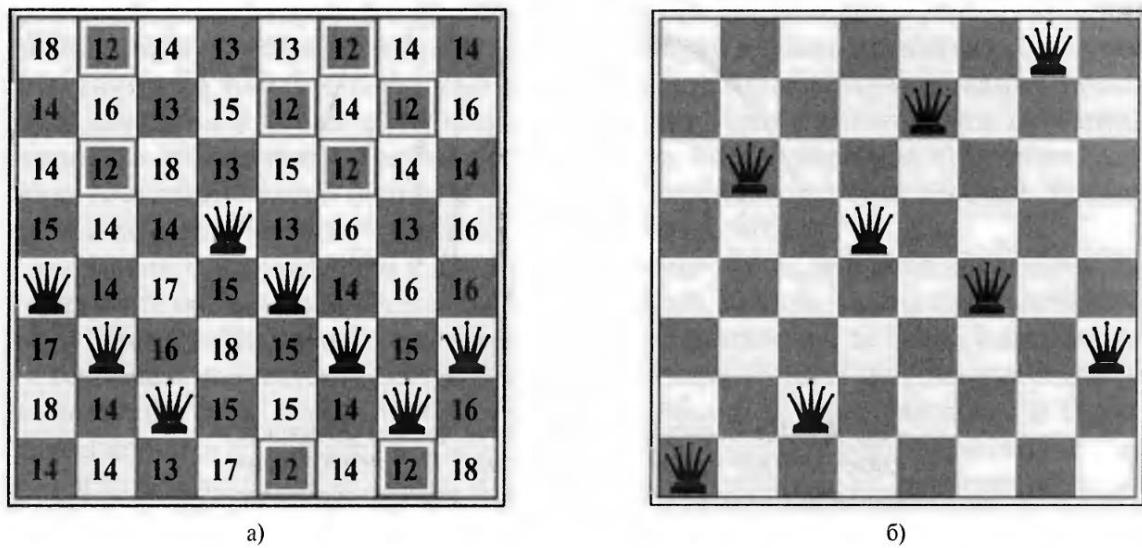


Рис. 4.8. Приклад застосування алгоритму зі сходженням до вершини: діаграма стану задачі з вісімома ферзями, що характеризується евристичною оцінкою вартості $h = 17$; на цій діаграмі показано значення h для кожного можливого наступника, отримане шляхом пересування ферзя в межах свого стовпця; відмічені найкращі ходи (а); локальний мінімум в просторі станів задачі з вісімома ферзями; цей стан має оцінку $h=1$, але кожен наступник характеризується більш високою вартістю (б)

На жаль, пошук зі сходженням до вершини часто заходить у безвихід по описаним нижче причинам.

- **Локальні максимуми.** Локальний максимум є піком, більш високим в порівнянні з кожним з його сусідніх станів, але нижчий, ніж глобальний максимум. Алгоритми зі сходженням до вершини, які досягають околиць локального максимуму, забезпечують просування вгору, до цього піку, але після цього заходять у безвихід, з якої більше нема куди рухатися. Така проблема схематично показана на рис. 4.7. Більш конкретний приклад полягає в тому, що стан, показаний на рис. 4.8, б, фактично є локальним максимумом (тобто локальний мінімум для оцінки вартості h); задача ще не розв'язана, а при будь-якому пересуванні окремо взятого ферзя ситуація стає ще гіршою.
- **Хребти.** Приклад хребта показаний на рис. 4.9. За наявності хребтів виникають послідовності локальних максимумів, задача проходження яких для жадібних алгоритмів є дуже важкою.
- **Плато.** Це область в ландшафті простору станів, в якій функція оцінки є плоскою. Воно може бути плоским локальним максимумом, з якого не існує виходу вгору, або **уступом**, з якого можливе подальше успішне просування (див. рис. 4.7). Пошук зі сходженням до вершини може виявитися нездатним вийти за межі плато.

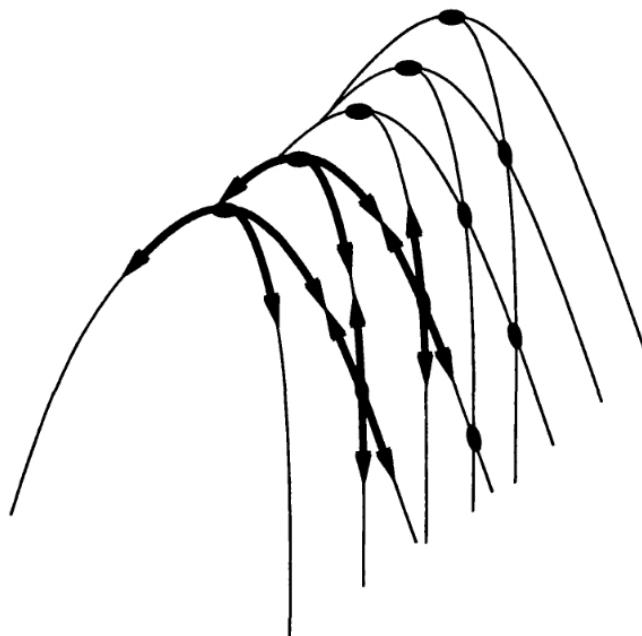


Рис. 4.9. Ілюстрація того, чому хребти викликають складнощі при сходженні до вершини. На хребет, що височіє зліва направо, накладаються грани станів (напівжирні дуги), створюючи послідовність локальних максимумів, які не є безпосередньо пов'язаними один з одним. У кожному локальному максимумі усі доступні дії спрямовані вниз.

В кожному з цих випадків даний алгоритм досягає такої точки, з якої не може здійснюватися подальше успішне просування. Починаючи з випадково

сформованого стану з вісімома ферзями, алгоритм пошуку зі сходженням до вершини по найкрутішому підйому заходить у безвихід в 86% випадках, вирішуючи тільки 14% екземплярів цієї задачі. Але він працює дуже швидко, виконуючи в середньому тільки 4 етапи у разі успішного завершення і 3 етапи, коли заходить у безвихід. Це не дуже поганий результат для простору станів з $8^8=17$ мільйонами станів.

Алгоритм, приведений в лістингу 4.2, зупиняється, коли досягнув плато, на якому найкращий наступник має таке ж значення, як і в поточному стані. Чи має сенс продовжувати рух, дозволивши **рух убік** сподіваючись на те, що це плато насправді є уступом, як показано на рис. 4.7? Відповідь на це питання зазвичай є позитивною, але необхідно дотримуватися обережності. Якщо буде завжди дозволено рух убік, притому що рух вгору неможливий, можуть виникати нескінченні цикли після того, як алгоритм досягне плоского локального максимуму, що не є уступом. Одно з широко вживаних рішень полягає в тому, що встановлюється межа кількості допустимих послідовних рухів убік. Наприклад, можна дозволити, припустимо, 100 послідовних рухів убік в завданні з вісімома ферзями. В результаті цього відносна кількість екземплярів завдання, що вирішуються за допомогою сходження до вершини, зростає з 14 до 94%. Але за цей успіх доводиться платити: алгоритм в середньому виконує приблизно 21 етап при кожному успішному розв'язку екземпляра задачі і 64 етапи при кожній невдачі.

Розроблено багато варіантів пошуку зі сходженням до вершини. При **стохастичному пошуку зі сходженням до вершини** здійснюється вибір випадковим чином одного з рухів вгору; ймовірність такого вибору може залежати від крутизни руху вгору. Зазвичай цей алгоритм сходить більш повільно в порівнянні з варіантом, що передбачає найбільш крутій підйом, але в деяких ландшафтах станів він знаходить найкращі рішення.

При **пошуку зі сходженням до вершини з вибором першого варіанту** реалізується стохастичний пошук зі сходженням до вершини шляхом формування наступників випадковим чином до тих пір, поки не буде

сформований наступник, кращий в порівнянні з поточним станом. Це – хороша стратегія, якщо будь-який стан має велику кількість наступників (вимірюється тисячами). У впр. 4.16 пропонується досліджувати цей алгоритм.

Алгоритми зі сходженням до вершини, описані вище, є неповними, оскільки часто виявляються нездатними знайти ціль, при тому, що вона існує, через те, що можуть зайди в глухий кут, досягнувши локального максимуму.

Пошук з сходженням до вершини і перезапуском випадковим чином керується широко відомої рекомендацією: "Якщо перша спроба виявилася невдалою, спробуйте знову і знову ". У цьому алгоритмі передбачено проведення ряду пошуків з сформованих випадковим чином початкових станів* і останов після досягнення цілі. Він є повним з ймовірністю, що досягає 1, навіть з тієї очевидної причини, що в ньому в кінцевому підсумку в якості початкового стану формується один з цільових станів.

Якщо ймовірність успіху кожного пошуку зі сходженням до вершини дорівнює p , то очікувана кількість необхідних перезапусків становить $1 / p$. Для примірників задачі з вісімома ферзями, якщо не дозволено рух вбік, $p \approx 0.14$, тому для знаходження цілі потрібно приблизно 7 ітерацій (6 невдалих і 1 успішна). Очікувана кількість етапів розв'язку дорівнює вартості однієї успішної ітерації, яка складається зі збільшеної в $(1-p) / p$ разів вартістю невдачі, або становить приблизно 22 етапів. Якщо дозволено рух убік, то в середньому потрібно $1 / 0.94 \approx 1.06$ ітерацій і $(1 \times 21) + (0.06 / 0.94) \times 64 \approx 25$ етапів. Тому алгоритм пошуку зі сходженням до вершини і перезапуском випадковим чином дійсно є дуже ефективним для задачі з вісімома ферзями. Навіть для варіанту з трьома мільйонами ферзів цей підхід дозволяє знаходити рішення менше ніж за хвилину**.

Успіх пошуку зі сходженням до вершини в значній мірі залежить від форми ландшафту простору станів: якщо в ньому є лише трохи локальних максимумів і плато, то пошук зі сходженням до вершини і перезапуском випадковим чином дозволяє дуже швидко знайти хороше рішення. З іншого

боку, багато реальних задач мають ландшафт, який більше нагадує сімейство дикобразів на плоскій підлозі, де на вершині голки кожного дикобраза живуть інші мініатюрні дикобрази і т.д., до нескінченності. NP-важкі завдання зазвичай мають експоненціальну кількість локальних максимумів, здатних завести алгоритм в глухий кут. Незважаючи на це, часто існує можливість знайти достатньо хороший локальний максимум після невеликої кількості перезапусків.

2.11.2 Пошук з емуляцією відпалу

Для будь-якого алгоритму зі сходженням до вершини, який ніколи не виконує руху "вниз по схилу", до станів з більш низькою оцінкою (або більш високою вартістю), гарантується, що він виявиться неповним, оскільки такий алгоритм завжди здатний зайти в глухий кут, досягнувши локального максимуму. На відміну від цього алгоритм з чисто випадковим блуканням (тобто з переміщенням до наступника, що обирається на рівних правах випадковим чином з множини наступників) є повним, але надзвичайно неефективним. Тому представляється розумної спроба скомбінувати якимось чином сходження до вершини з випадковим блуканням, що дозволить забезпечити і ефективність, і повноту. Алгоритмом такого типу є алгоритм з **емуляцією відпалу**. У металургії **відпалом** називається процес, що застосовують для відпуску металу і скла шляхом нагрівання цих матеріалів до високої температури, а потім поступового охолодження, що дозволяє перевести оброблюваний матеріал в низькоенергетичний кристалічний стан. Щоб зрозуміти суть емуляції відпалу, переведемо нашу увагу зі сходження до вершини на **градієнтний спуск** (тобто мінімізацію вартості) і уявімо собі, що наше завдання – загнати тенісну кульку в найглибшу лунку на нерівній поверхні. Якби ми просто дозволили кульці котитися по цій поверхні, то вона застрягла би в одному з локальних мінімумів. А струшуючи поверхню, можна виштовхнути кульку з локального мінімуму. Весь секрет полягає в тому, що поверхню потрібно трясти досить сильно, щоб кульку можна було виштовхнути з локальних мінімумів, але не настільки сильно, щоб вона вилетіла з глобального мінімуму. Процес пошуку розв'язку з емуляцією відпалу полягає в

тому, що спочатку відбувається інтенсивне струшування (аналогічне нагріванню до високої температури), після чого інтенсивність струшування поступово зменшується (що можна порівняти зі зниженням температури).

Найбільш внутрішній цикл алгоритму з емуляцією відпалу (лістинг 4.3) повністю аналогічний циклу алгоритму зі сходженням до вершини, але в ньому замість найкращого ходу виконується випадково обраний хід. Якщо цей хід покращує ситуацію, то завжди приймається. В іншому випадку алгоритм приймає даний хід з деякою ймовірністю, меншою за 1. Ця ймовірність зменшується експоненціально з "погіршенням" ходу – залежно від величини ΔE , на яку погіршується його оцінка. Крім того, ймовірність зменшується в міру зниження "температури" Т. "Погані" ходи швидше за все можуть бути дозволені на початку, коли температура висока, але стають менш ймовірними в міру зниження Т. Можна довести, що якщо в графіку *schedule* передбачено досить повільне зниження Т, то даний алгоритм дозволяє знайти глобальний оптимум з ймовірністю, що наближається до 1.

На початку 1980-х років, пошук з емуляцією відпалу широко використовувався для вирішення завдань компонування НВІС. Крім того, цей алгоритм знайшов широке застосування при вирішенні задач планування виробництва й інших великомасштабних завдань оптимізації. У впр. 4.16 пропонується порівняти його продуктивність з продуктивністю пошуку зі сходженням до вершини і перезапуском випадковим чином при вирішенні задачі з п ферзями.

Лістинг 4.3 . Алгоритм пошуку з емуляцією відпалу, який є однією з версій алгоритму стохастичного пошуку зі сходженням до вершини, в якому дозволені деякі ходи вниз. Ходи вниз приймаються до виконання з більшою ймовірністю на ранніх етапах виконання графіка відпалу, а потім, по мірі того як проходить час, виконуються менш часто. Вхідний параметр *schedule* визначає значення температури Т як функції від часу

function Simulated-Annealing(*problem*, *schedule*) **returns** стан розв'язку

inputs: *problem*, задача

schedule, відображення між часом і "температуру"

local variables: current, вузол

next, вузол

T, "температура", від якої залежить ймовірність кроків

ВНИЗ

current \leftarrow Make-Node(Initial-State[problem])

for t \leftarrow 1 **to** ∞ **do**

T \leftarrow schedule[t]

if T = 0 **then return** current

next \leftarrow випадково обраний наступник стану current

$\Delta E \leftarrow$ Value[next] - Value[current]

if $\Delta E > 0$ **then** current \leftarrow next

else current \leftarrow next з ймовірністю тільки $e^{\Delta E/T}$

2.11.3 Локальний променевий пошук

Прагнення подолати обмеження, пов'язані з нестачею пам'яті, призвело до того, що свого часу перевага віддавалася алгоритмам, що передбачають зберігання в пам'яті тільки одного вузла, але, як виявилося, такий підхід часто є занадто радикальним способом економії пам'яті. В алгоритмі **локального променевого пошуку***** передбачено відстеження k станів, а не тільки одного стану. Робота цього алгоритму починається з формування випадковим чином k станів. На кожному етапі формуються всі наступники всіх k станів. Якщо будь-який з цих наступників відповідає цільовому стану, алгоритм зупиняється. В іншому випадку алгоритм обирає із загального списку k найкращих наступників і повторює цикл.

На перший погляд може здатися, що локальний променевий пошук з k станами є ні чим іншим, як виконанням k перезапусків випадковим чином, але не послідовно, а паралельно. Проте в дійсності ці два алгоритми є повністю

різними. При пошуку з перезапуском випадковим чином кожен процес пошуку здійснюється незалежно від інших. *А в локальному променевому пошуку корисна інформація передається по k паралельних потоках пошуку.* Наприклад, якщо в одному стані виробляється кілька хороших наступників, а у всіх інших $k-1$ станах виробляються погані наступники, то виникає такий ефект, як якщо би потік, контролюючий перший стан, повідомив іншим потокам: "Ідіть всі сюди, тут трава зеленіша!" Цей алгоритм здатний швидко відмовитися від безплідних пошуків і перекинути свої ресурси туди, де досягнутий найбільший прогрес.

У своїй найпростішій формі локальний променевий пошук може страждати від відсутності різноманітності між к станами, оскільки всі ці стани здатні швидко зосередитися в невеликому регіоні простору станів, в результаті чого цей пошук починає ненабагато відрізнятися від дорогої версії пошуку зі сходженням до вершини. Цей недолік дозволяє усунути варіант, що називають **стохастичним променевим пошуком**, який аналогічний стохастичному пошуку зі сходженням до вершини. При стохастичному променевому пошуку замість вибору найкращих к наступників з пулу наступників-кандидатів відбувається вибір k наступників випадковим чином, при тому що ймовірність вибору даного конкретного наступника є зростаючою функцією значення його оцінки. Стохастичний променевий пошук має деяку схожість з процесом природного відбору, в якому "наступники" (нащадки) деякого "стану" (організму) утворюють наступне покоління відповідно до "значення їх оцінки" (відповідно до їх життєвої придатності).

* Може виявитися важкою навіть сама задача формування випадковим чином деякого стану з неявно заданого простору станів.

** В [958] доведено, що в деяких випадках краще за все здійснювати перезапуск рандомізованого алгоритму пошуку після закінчення конкретної,

постійної тривалості часу і що такий підхід може виявитися набагато більш ефективним в порівнянні з тим, коли дозволено продовжувати кожний пошук до нескінченності. Прикладом такого підходу є також заборона або обмеження кількості рухів в бік.

*** Локальний променевий пошук є адаптацією променевого пошуку, який є алгоритмом, заснованим на використанні шляху.

2.12 Задачі виконання обмежень

В цьому розділі показано, що розглядаючи стани на більш високому рівні деталізації, ніж просто як маленькі «чорні ящики», можна прийти до створення цілого ряду потужних нових методів пошуку і більш глибокому розумінню структури і складності задачі.

В розділах 3 і 4 розглядався підхід, згідно до якого задачі можна розв'язувати, виконуючи пошук в просторі **станів**. Для реалізації цього підходу стани необхідно оцінювати з допомогою евристичних функцій, відповідних даний проблемній області, а також перевіряти для визначення того, чи не є вони цільовими станами. Однак с точки зору таких алгоритмів пошуку кожний стан представляє собою **чорний ящик** з нерозрізеною внутрішньою структурою. Вони представлені з допомогою довільно вибраної структури даних, доступ до якої може здійснюватися лише з використанням процедур, що відносяться до даної проблемної області, - функції визначення спадкоємця, евристичної функції і процедури перевірки цілі.

В даному розділі розглядаються **задачі виконання обмежень**, в яких стани і перевірка цілі відповідають стандартному, структурованому і дуже простому **представленню** (див. розділ 5.1). Це дозволяє визначати алгоритми пошуку, здатні скористатись перевагою такої структури станів, і використати для розв'язання великих задач евристичні функції загального призначення, а не функції, які відносяться до конкретної задачі (див. розділи 5.2, 5.3). Але, можливо, ще більше важливо те, що використовуване стандартне представлення перевірки цілі розкриває структуру самої задачі (див. розділ 5.4).

Це дозволяє створити методи декомпозиції задачі й зрозуміти внутрішній зв'язок між структурою задачі та складністю її розв'язку.

2.12.1 Задачі виконання обмежень

Формально кажучи, **будь-яка задача виконання обмежень** (Constraint Satisfaction Problem – CSP) визначена множиною **змінних**, x_1, x_2, \dots, x_n , і множиною **обмежень**, c_1, c_2, \dots, c_m . Кожна змінна x_i має непусту **область визначення** D_i можливих **значень**. Кожне обмеження C_i включає деяку підмножину змінних і задає допустимі комбінації значень для цієї підмножини. Стан задачі визначається шляхом **присвоювання** значень деяким або всім цим змінним, $\{x_i = v_i, x_j = v_j, \dots\}$. Присвоєння, яке не порушує ніяких обмежень, називається **сумісним**, або допустимим присвоюванням. Повним називається таке присвоєння, в якому бере участь кожна змінна, а **рішенням** задачі CSP є повне присвоєння, яке виконує всі обмеження. Крім того, для деяких задач CSP необхідно знайти рішення, яке максимізує **цільову функцію**.

Як же весь цей опис перевести на практику? Припустимо, що втомившись від Румунії, ми розглядаємо карту Австралії, на котрій показані всі її штати і території (рис. 5.1, а), і що ми отримали завдання розфарбувати кожен регіон в червоний, зелений або синій колір таким чином, щоб жодна пара сусідніх регіонів не мала однакового кольору. Щоб сформулювати це завдання у вигляді задачі CSP, означимо в якості змінних скорочені позначення цих регіонів: WA, NT, Q, NSW, V, SA і T. Областю визначення кожної змінної є множина кольорів {red, green, blue}. Обмеження потребують, аби всі пари сусідніх регіонів мали різні кольори; наприклад, допустимими комбінаціями для WA і NT є наступні пари:

$$\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$$

(Це обмеження можна також представити більш стисло у вигляді нерівності $WA \neq NT$, за умови, що в даному алгоритмі виконання обмежень передбачений деякий спосіб обчислення таких виразів.) Кількість можливих розв'язків достатньо велика; наприклад, одним із таких розв'язків є наступний:

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red\}$$

Задачу CSP зручно представити візуально у вигляді **графа обмежень**, як показано на рис. 5.1, б. Вузли цього графа відповідають змінним задачі, а дуги – обмеженням.

Розглядаючи деяку задачу у вигляді задачі CSP, можна досягнути декількох важливих переваг. Представлення станів в задачі CSP відповідає деякому стандартному шаблону (тобто виражається у вигляді множини змінних з присвоєними значеннями), тому функцію визначення нащадку і перевірку цілі можна записати в універсальній формі, що може бути застосована до всіх задач CSP. Більш того, можуть бути розроблені ефективні, універсальні евристичні функції, для створення яких не потребується додаткових знань про конкретну проблемну область. Нарешті, для спрощення процесу розв’язку може використовуватись сама структура графа обмежень, що дозволяє в деяких випадках досягти експоненціального зменшення складності. Це представлення задачі CSP є першим і найбільш простим з ряду схем представлення, які будуть розроблятись протягом цієї книги.

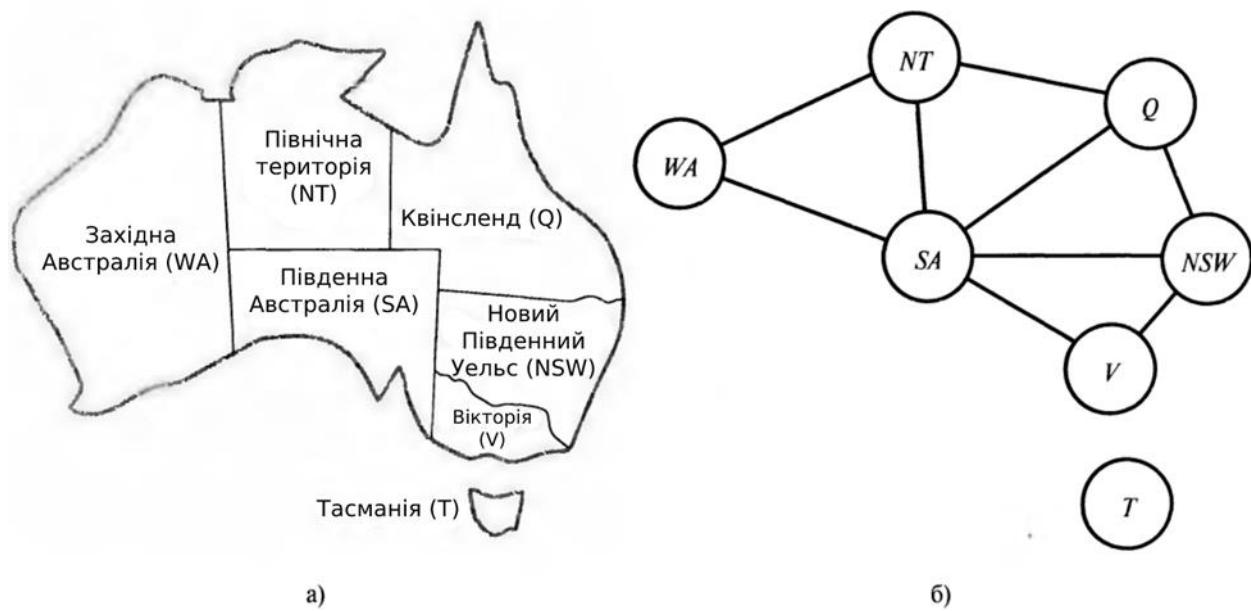


Рис. 5.1. Приклад постановки задачі CSP: основні штати і території на карті Австралії (а); розфарбовка цієї карти може розглядатись як задача виконання обмежень; завдання полягає в тому, щоб назначити колір кожному регіону так, аби жодна пара сусідніх регіонів не мала того самого кольору;

розв'язувана задача розфарбування карти, представлена у вигляді графа обмежень

(6)

Можна доволі легко показати, що будь-якій задачі CSP може бути дане **інкрементне формулювання**, як і будь-якій стандартній задачі пошуку, наступним чином.

- **Початковий стан.** Пусте присвоєння $\{ \}$, в якому жодній змінній не присвоєно значення.
- **Функція визначення нащадку.** Значення може бути присвоєно будь-якій змінній з неприсвоєним значенням, за умови, що змінна не буде конфліктувати з іншими змінними, значення яким були присвоєні раніше.
- **Перевірка цілі.** Поточне присвоєння є повним.
- **Вартість шляху.** Постійна вартість (наприклад, 1) для кожного етапу.

Кожен розв'язок повинен бути повним присвоєнням и тому має знаходитись на глибині n , якщо наявні n змінних. Крім того, дерево пошуку розповсюджується лише на глибину n . Через ці причини для розв'язання задач CSP широко використовуються алгоритми пошуку в глибину (див. розділ 5.2). До того ж такі задачі характерні тим, що сам шлях, по котрому досягається деякий розв'язок, не є цікавим. Тому може також використовуватись **постановка з повним станом**, в якій кожен стан являє собою повне присвоєння, що виконує або не виконує задані обмеження. На основі такої постановки гарно діють методи локального пошуку (див. розділ 5.3.)

Задачі CSP найпростішого вигляду характеризуються тим, що в них використовуються змінні, які є **дискретними і мають скінченні області визначення**. До такого вигляду відноситься задача розфарбування карти. Задача про вісім ферзів, що описана в розділі 3, також може розглядатись як задача CSP зі скінченною областю визначення, де змінні Q_1, \dots, Q_8 позначають позиції кожного ферзя на стовбцях 1, ..., 8, а кожна змінна має область визначення $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Якщо максимальний розмір області визначення будь-якої змінної в задачі CSP рівний d , то кількість можливих повних присвоєнь вимірюється величиною $O(d^n)$, тобто залежить експоненціально від кількості змінних. До категорії задач CSP зі скінченою областю визначення відносяться **булеві задачі CSP**, в яких змінні можуть мати

значення або true, або false. Булеві задачі CSP включають в якості часткових випадків деякі NP-повні задачі, такі як 3SAT (див. розділ 7). Тому в найгіршому випадку не можна розраховувати на те, що ми зможемо розв'язувати задачі CSP зі скінченою областю визначення за час, що менше експоненціального. Але в більшості практичних застосувань алгоритми CSP загального призначення дозволяють розв'язувати задачі, на кілька порядків величини більші у порівнянні з тими, які можуть бути розв'язані з допомогою алгоритмів пошуку загального призначення, представлених у розділі 3.

Дискретні змінні можуть також мати нескінченні області визначення, наприклад, такі, як множина всіх цілих чисел або множина всіх рядків. Зокрема, при календарному плануванні будівельних робіт дата початку кожної роботи є змінною, а її можливими значеннями є ціличисельні інтервали часу в добі, що рахуються від поточної дати. При розв'язанні задач з нескінченними областями визначення більше нема можливостей описувати обмеження, перераховуючи всі допустимі комбінації значень. Замість цього має використовуватись **мова обмежень**. Наприклад, якщо робота Job_1 , яка займає п'ять днів, повинна передувати роботі Job_3 , то для опису цієї умови потрібна мова обмежень, представлених у вигляді алгебраїчних нерівностей, таких як $StartJob_1 + 5 \leq StartJob_3$. Крім того, більше нема можливостей розв'язувати такі обмеження, лише перераховуючи всі можливо присвоєння, оскільки кількість подібних можливих присвоєнь нескінченно велика. Для **лінійних обмежень** з ціличисельними змінними (тобто обмежень, подібних щойно наведеному, в яких кожна змінна представлена лише в лінійній формі) існують спеціальні алгоритми пошуку розв'язків (які тут не розглядаються). Можна довести, що не існує алгоритмів розв'язання загальних **нелінійних обмежень** з ціличисельними змінними. В деяких випадках задачі з ціличисельними обмеженнями можна звести до задач зі скінченою областю визначення, встановлюючи границі значень всіх цих змінних. Наприклад, в задачі планування можна встановити верхню границю, рівну загальній тривалості всіх робіт, які мають бути заплановані.

В реальному світі дуже часто зустрічаються задачі виконання обмежень з **неперервними областями визначення**, і ці задачі інтенсивно вивчаються в області дослідження операцій. Наприклад, для планування експериментів на космічному телескопі Габбл потребується дуже точна прив'язка спостережень по часу; початок і кінець кожного спостереження і кожного маневру є двома змінними зі значеннями із неперервної області визначення, які повинні підпорядковуватись всіляким астрономічним обмеженням, обмеженням передування і обмеженням потужності двигунів. Одною з широко відомих категорій задач CSP з неперервною областю визначення є задачі **лінійного програмування**, в яких обмеження повинні являти собою лінійні нерівності, що утворюють опуклу область. Задачі лінійного програмування можуть бути розв'язані за час, який залежить поліноміально від кількості змінних. Крім того, проводились дослідження задач з іншими типами обмежень і цільових функцій – задачі квадратичного програмування, конічного програмування другого порядку і т.д.

Крім дослідження типів змінних, які можуть бути присутні в задачах CSP, корисно зайнятись вивченням типів обмежень. Найпростішим типом обмеження є **унарне обмеження**, яке обмежує значення одної змінної. Наприклад, може виявитись, що жителі штату Південна Австралія дуже не люблять зелений колір, {green}. Кожне унарне обмеження можна усунути, виконуючи попередню обробку області визначення відповідної змінної, аби видалити будь-яке значення, що порушує це обмеження. **Бінарне обмеження** пов'язує між собою дві змінні. Наприклад, бінарним обмеженням є $SA \neq NSW$. Бінарною задачею CSP називається задача, в якій передбачені виключно бінарні обмеження; вона може бути представлена у вигляді графа відношень, подібного наведеному на рис. 5.1, б.

В обмеженнях високого порядку беруть участь три або більше змінних. Одним із відомих прикладів таких задач є **криптоарифметичні** головоломки, що також звуться числовими ребусами (рис. 5.2, а). Зазвичай висувається вимога, аби кожна буква в криптоарифметичній головоломці була окремою цифрою. В випадку задачі, показаної на рис. 5.2, а, така вимога може бути

виражена за допомогою обмеження з шести змінними $\text{Alldiff}(F, T, U, W, R, O)$. Іншим чином ця вимога може бути представлена у вигляді колекції бінарних відношень, таких як $F \neq T$. Обмеження складання для чотирьох стовпців цієї головоломки також включають декілька змінних і можуть бути записані наступним чином:

$$O + O = R + 10 \cdot X_1$$

$$X_1 + W + W = U + 10 \cdot X_2$$

$$X_2 + T + T = O + 10 \cdot X_3$$

$$X_3 = F$$

Де X_1, X_2 та X_3 – **допоміжні змінні**, що являють собою цифру (0 або 1), яка переноситься до наступного стовпця. Обмеження високого порядку можуть бути представлені у вигляді **гіперграфу обмежень**, подібного наведеному на рису. 5.2, б. Уважний читач помітить, що обмеження *Alldiff* може бути розбито на бінарні обмеження - $F \neq T, F \neq U$ і т.д. Справді, у впр. 5.11 пропонується довести, що кожне обмеження високого порядку зі скінченною областю визначення можна звести до деякої кількості бінарних обмежень, вводячи достатню кількість допоміжних змінних. У зв'язку з цим в даному розділі будуть розглядатись лише бінарні відношення.

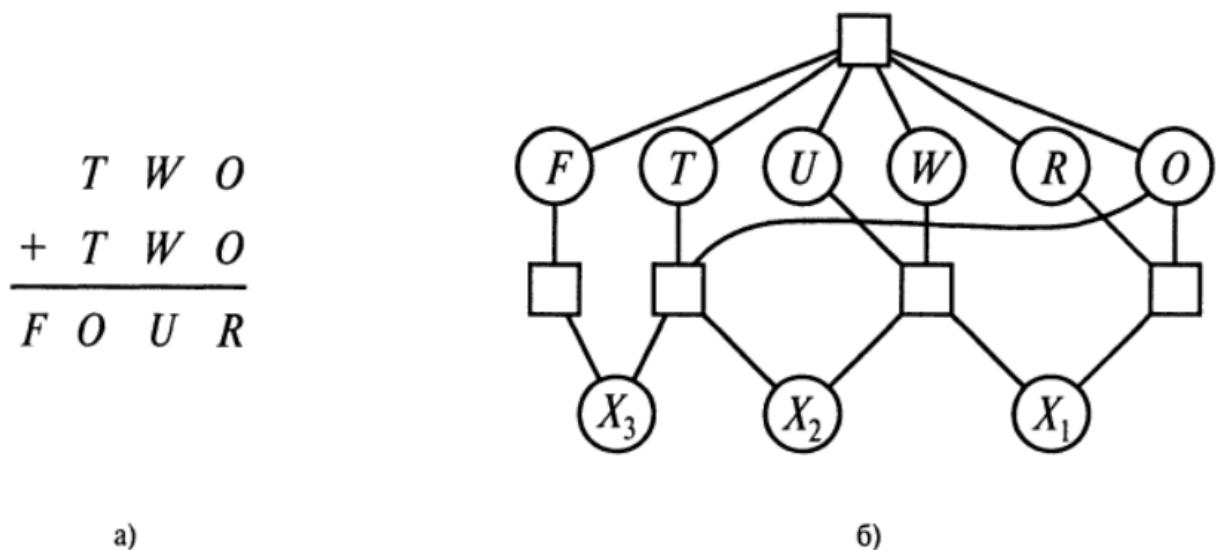


Рис. 5.2. Приклад криптоарифметичної задачі; кожна літера позначає окрему цифру; завдання полягає в тому, аби знайти таку заміну літер

цифрами, аби результатуючий вираз сумування був арифметично правильним, з додатковим обмеженням, що наявність ведучих нулів не допускається (а); гіперграф обмежень для цієї криптоарифметичної задачі, на якому показано обмеження Alldiff, а також обмеження сумування по стовпцям; кожне обмеження позначено квадратом, який поєднаний з обмежуваними їм змінними (б)

Всі обмеження, що розглядалися до сих пір, були **абсолютними** обмеженнями, порушення яких рівносильно тому, що якийсь потенціальний розв'язок більше не розглядається як такий. З іншого боку, в багатьох реальних задачах CSP застосовуються обмеження **переваги**, які вказують, якому розв'язку надати перевагу. Наприклад, в задачі складання розкладу занять в університеті може знадобитися врахувати, що професор X бажає проводити заняття зранку, а професор Y – пообіді. Розклад занять, в якому передбачено проведенням професором X заняття в 14:00, все ще може розглядатись як розв'язок (якщо не виявиться, що професор X повинен в цей час завідувати засіданням кафедри), але вже не буде оптимальним. Обмеження переваги часто можна закодувати як вартості присвоювання окремим змінним; наприклад, за призначення професору X пообіднього інтервалу доведеться заплатити 2 пункти, які будуть враховані в сумарному значенні цільової функції, в той час як ранковий інтервал має вартість 1 пункт. При використанні такої постановки задачі CSP з перевагами можна розв'язувати, використовуючи методи пошуку з оптимізацією, або основані на пошуку шляху, або локальні. Подібні задачі CSP в даному розділі більше не розглядаються, але в розділі з бібліографічними нотатками наведені деякі посилання.

2.12.2 Застосування пошуку з поверненням для розв'язання задач CSP

У попередньому розділі приведена постановка задач CSP у вигляді задач пошуку. Якщо використовується така постановка, то з'являється можливість розв'язувати задачі CSP з допомогою будь-яких алгоритмів пошуку, наведених в розділах 3 і 4. Припустимо, що ми застосовуємо алгоритм пошуку в ширину до універсальної постановки задачі CSP, наведеної в попередньому розділі. Але

ми швидко помітимо дещо жахливе: коефіцієнт розгалуження на верхньому рівні дорівнює nd , оскільки будь-яке із d значень може бути присвоєно будь-якій з n змінних. На наступному рівні коефіцієнт розгалуження дорівнює $(n - 1)d$ і т.д., на всіх n рівнях. При цьому створюється дерево з $n! \cdot d^n$ гілками, навіть не зважаючи на те, що є лише d^n можливих повних присвоєнь!

В застосованій нами постановці задачі, яка зовні здавалась розумною, але виявилась погано продуманою, не було враховано істотно важливу властивість, спільну для всіх задач CSP – **комутативність**. Задача називається комутативною, якщо порядок виконання будь-якої конкретної послідовності дій в процесі її розв'язання не впливає на результат. Саме ця властивість характерна для задач CSP, оскільки, присвоюючи значення змінним, ми досягаємо одного і того ж часткового присвоєння незалежно від порядку присвоєння. Тому в усіх алгоритмах пошуку CSP нашадки формуються враховуючи можливі присвоєння тільки для єдиної змінної в кожному вузлі дерева пошуку. Наприклад, у кореневому вузлі дерева пошуку в задачі розфарбування мапи Австралії можна мати вибір між $SA = red, SA = green$ і $SA = blue$, але ми ніколи не повинні обирати між $SA = red$ та $WA = blue$. Визначивши таку умову, можна сподіватись зменшити кількість листів до d^n .

Пошук в глибину, в якому кожен раз обираються значення для однієї змінної і виконується повернення, якщо більше не залишається допустимих значень, які можна було би присвоїти змінній, називається **пошуком з поверненнями**. Цей алгоритм пошуку наведений в лістингу 5.1. Зверніть увагу на те, що в цьому алгоритмі по суті використовується метод інкрементного формування нашадків по одному за один раз, який описаний на с. 131. Крім того, для формування нашадку поточне присвоєння доповнюється, а не копіюється. Оскільки це представлення задач CSP стандартизовано, то у функції Backtracking-Search не потребується враховувати початковий стан, функцію визначення нашадку або перевірку цілі, характерні для розглядуваної проблемної області. Частина дерева пошуку для задачі розфарбування мапи

Австралії показана на рис. 5.3, де значення присвоюються змінним у порядку WA, NT, Q, \dots .

Лістинг 5.1. Простий алгоритм пошуку з поверненнями для розв'язання задач виконання обмежень csp. Цей алгоритм складено за принципом рекурсивного пошуку в глибину, описаного в розділі 3. Функції Select-Unassigned-Variable і Order-Domain-Values можуть використовуватись для реалізації евристичних функцій загального призначення, які розглядаються в тексті даного розділу.

function Backtracking-Search (csp) **returns** розв'язок result або індикатор відмови failure

return Recursive-Backtracking ({ }, csp)

function Recursive-Backtracking (assignment, csp) **returns** розв'язок result або індикатор відмови failure

if присвоєння assignment є повним **then return** assignment

var \leftarrow Select-Unassigned-Variable (Variables[csp], assignment, csp)

for each in value in Order-Domain-Values(var, assignment, csp) **do**

if значення value є сумісним з присвоєнням assignment
відповідно обмежень Constraints[csp] **then**

додати { var = value } до присвоєння assignment

result \leftarrow Recursive-Backtracking (assignment, csp)

if result \neq failure **then return** result

видалити { var = value } з присвоєння assignment

return failure

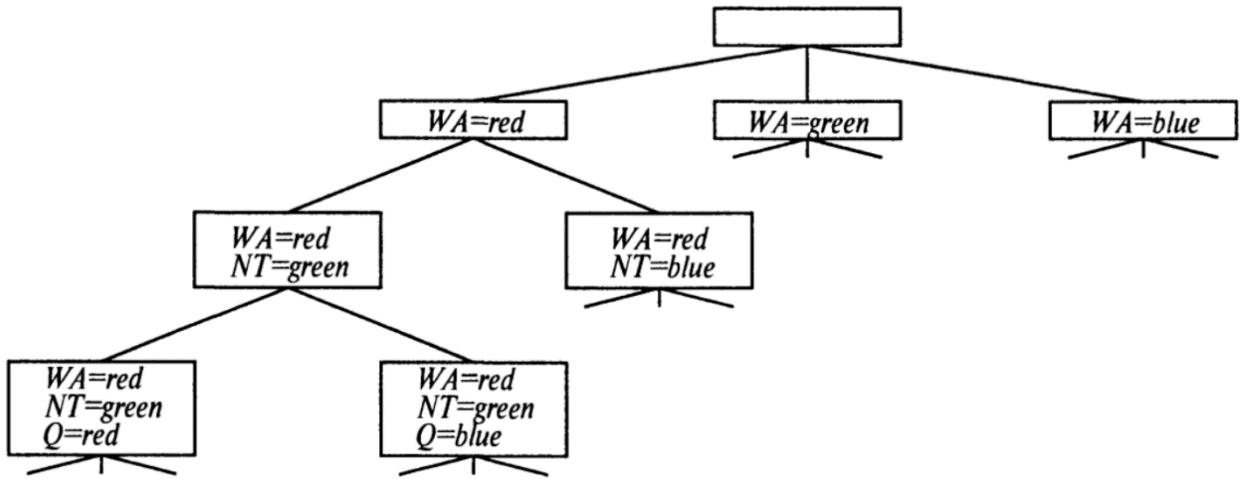


Рис 5.3. Частина дерева пошуку, сформованого шляхом простого пошуку з поверненнями при розв'язанні задачі розфарбування мати, наведеної на рис. 5.1.

Згідно означенням, наведеним в розділі 3, алгоритм простого пошуку з поверненнями являє собою неінформативний алгоритм, тому не варто розраховувати нате, що він виявиться дуже ефективним при розв'язанні складних задач. Результати його використання до деяких прикладів задач показані у першому стовпці табл. 5.1 і підтверджують ці припущення.

В розділі 4 було показано, що такий недолік неінформованих алгоритмів пошуку, як низька продуктивність, можна усунути, передбачивши використання в них евристичних функцій, відповідних даній проблемній області, які засновані на наших знаннях про конкретну задачу. Як виявилося, задачі CSP можна розв'язувати ефективно без таких знань про конкретну проблемну область. Замість цього в даному розділі будуть розроблені методи загального призначення, що дозволяють знайти відповідь на перераховані питання:

- Який змінній повинно бути присвоєно значення в наступну чергу і в якому порядку необхідно намагатись присвоїти ці значення?
- Як впливають поточні присвоєння значень змінним на інші змінні з неприсвоєними значеннями?
- Якщо якийсь шлях виявився невдалим (тобто досягнуто стан, в якому жодна змінна не має допустимих значень), чи дозволяє пошук уникнути повтору цієї невдачі при проходженні наступних шляхів?

В наведених нижче підрозділах по черзі дані відповіді на кожне з цих питань.

Таблиця 5.1. Результати застосування різних алгоритмів CSP для розв'язування різних задач. Зліва направо показані алгоритми простого пошуку з поверненнями, пошуку з поверненнями на основі евристичної функції MRV, локального пошуку з попередньою перевіркою, локального пошуку з попередньою перевіркою на основі MRV і локального пошуку з мінімальними конфліктами. В кожній комірці показана середня кількість перевірок сумісності (за п'ять прогонів), які знадобились для розв'язання даної задачі; зверніть увагу на те, що всі ці числа, крім двох, що знаходяться зверху праворуч, виражені в тисячах (К). Числа в круглих дужках показують, що за зазначену кількість перевірок не було знайдено жодного розв'язку. Першою задачею є пошук розфарбування в 4 кольорі для 50 штатів США. Решта задач взяті із [56, табл. 1]. В другій задачі підраховується загальна кількість перевірок, необхідних для розв'язання всіх задач з n ферзями для n від 2 до 50. Третью задачею є «головоломка із зеброю», що описана у впр. 5.13. Останні дві задачі являють собою штучні задачі, що складаються випадковим чином (алгоритм з мінімальними конфліктами до них не застосовувався). Ці результати показують, що попередня перевірка на основі евристичної функції MRV є найкращим способом розв'язання всіх цих задач порівняно з іншими алгоритмами пошуку з поверненнями, але цей метод не завжди випереджає локальний пошук з мінімальними конфліктами.

Задача	Пошук з поверненнями	Пошук з поверненнями на основі MRV	Локальний пошук з попередньою перевіркою	Локальний пошук з попередньою перевіркою на основі MRV	Локальний пошук з мінімальними конфліктами
Розфарбування в 4 кольорі	10000	10000	10000	10000	10000
Задача 1	10000	10000	10000	10000	10000
Задача 2	10000	10000	10000	10000	10000
Задача 3	10000	10000	10000	10000	10000
Задача 4	10000	10000	10000	10000	10000
Задача 5	10000	10000	10000	10000	10000
Задача 6	10000	10000	10000	10000	10000
Задача 7	10000	10000	10000	10000	10000
Задача 8	10000	10000	10000	10000	10000
Задача 9	10000	10000	10000	10000	10000
Задача 10	10000	10000	10000	10000	10000
Задача 11	10000	10000	10000	10000	10000
Задача 12	10000	10000	10000	10000	10000
Задача 13	10000	10000	10000	10000	10000
Задача 14	10000	10000	10000	10000	10000
Задача 15	10000	10000	10000	10000	10000
Задача 16	10000	10000	10000	10000	10000
Задача 17	10000	10000	10000	10000	10000
Задача 18	10000	10000	10000	10000	10000
Задача 19	10000	10000	10000	10000	10000
Задача 20	10000	10000	10000	10000	10000
Задача 21	10000	10000	10000	10000	10000
Задача 22	10000	10000	10000	10000	10000
Задача 23	10000	10000	10000	10000	10000
Задача 24	10000	10000	10000	10000	10000
Задача 25	10000	10000	10000	10000	10000
Задача 26	10000	10000	10000	10000	10000
Задача 27	10000	10000	10000	10000	10000
Задача 28	10000	10000	10000	10000	10000
Задача 29	10000	10000	10000	10000	10000
Задача 30	10000	10000	10000	10000	10000
Задача 31	10000	10000	10000	10000	10000
Задача 32	10000	10000	10000	10000	10000
Задача 33	10000	10000	10000	10000	10000
Задача 34	10000	10000	10000	10000	10000
Задача 35	10000	10000	10000	10000	10000
Задача 36	10000	10000	10000	10000	10000
Задача 37	10000	10000	10000	10000	10000
Задача 38	10000	10000	10000	10000	10000
Задача 39	10000	10000	10000	10000	10000
Задача 40	10000	10000	10000	10000	10000
Задача 41	10000	10000	10000	10000	10000
Задача 42	10000	10000	10000	10000	10000
Задача 43	10000	10000	10000	10000	10000
Задача 44	10000	10000	10000	10000	10000
Задача 45	10000	10000	10000	10000	10000
Задача 46	10000	10000	10000	10000	10000
Задача 47	10000	10000	10000	10000	10000
Задача 48	10000	10000	10000	10000	10000
Задача 49	10000	10000	10000	10000	10000
Задача 50	10000	10000	10000	10000	10000

Розфарбування в 4 кольори для 50 штатів США	(> 1000K)	(>1000K)	2K	60	64
Задача з піферзями	(>40 000K)	13 500K	(>40 000K)	817K	4K
Головоломка з зеброю	3859K	1K	35K	0.5K	2K
Випадкова задача 1	415K	3K	26K	2K	-
Випадкова задача 2	942K	27K	77K	15K	-

2.12.3 Упорядкування змінних та значень

Наведений вище алгоритм пошуку з поверненнями містить наступний рядок: $\text{var} \leftarrow \text{Select-Unassigned-Variable}(\text{Variables}[csp], \text{assignment}, csp)$

За замовченнем функція `Select-Unassigned-Variable` обирає наступну змінну з неприсвоєним значенням у порядку, вказаному у списку `Variables[csp]`. Таке статичне упорядкування змінних рідко призводить до найбільш ефективного пошуку. Наприклад, після присвоювань $WA = red$ і $NT = green$ залишається лише одне можливе значення для SA , тому має сенс на наступному етапі виконувати присвоєння $SA = blue$, а не присвоювати значення змінній Q . Насправді після присвоєння значення змінній SA всі варіанти вибору значень для Q , NSW та V стають вимушеними. Ця інтуїтивна ідея (згідно якої в першу чергу варто обирати змінну з найменшою кількістю «допустимих» значень) називається евристикою з мінімальною кількістю значень, що залишились (Minimum Remaining Values - MRV). Цю евристику називають також евристикою зі «змінною, на яку розповсюджується найбільша кількість

обмежень», або евристикою «першого невдалого завершення»; остання назва застосовується через те, що така евристична функція передбачає вибір змінної, яка з найбільшою вірогідністю з часом призведе до невдачі, усікаючи тим самим дерево пошуку. Якщо існує змінна x з нульовою кількістю залишених допустимих значень, евристична функція MRV вибере x і невдача буде виявлена миттєво; це дозволяє запобігти безглупих пошуків серед інших змінних, які завжди будуть закінчуватись невдачею після того, як буде обрана змінна x . Продуктивність пошуку з використанням цієї евристичної функції показана в другому стовпці табл. 5.1, позначеному як Пошук з поверненнями на основі MRV. Продуктивність пошуку підвищується від 3 до 3000 разів порівняно зі звичайним пошуком з поверненнями, в залежності від задачі. Варто відмітити, що в використаному тут показнику продуктивності не враховується додаткова вартість обчислення значень евристичної функції; в наступному розділі описано метод, який дозволяє утримувати значення вартості в допустимих границях.

Ця степенева функція MRV взагалі не надає ніякої допомоги при виборі розфарбування в Австралії першого регіону, оскільки початково кожен регіон має три допустимих кольори. В такому випадку зручною стає **степенева евристика**. В цій евристиці робиться спроба зменшити коефіцієнт розгалуження в майбутніх варіантах за рахунок вибору змінної, яка бере участь в найбільшій кількості обмежень на інші змінні з неприсвоєними значеннями. На рис. 5. 1 змінною з найбільшим степенем, 5, є змінна SA; інші змінні мають степінь 2 або 3, окрім T, яка має степінь 0. Насправді після вибору змінної SA використання степеневої евристики дозволяє розв'язати задачу без будь-яких невдалих етапів – тепер можна вибрати будь-який сумісний колір в кожній точці вибору і все одно прийти до рішення без пошуку з поверненнями. Евристика з мінімальною кількістю залишкових значень зазвичай забезпечує більш потужне керівництво, але степенева евристика може виявитись корисною в якості засобу розв'язання невизначених ситуацій.

Після вибору однієї зі змінних в цьому алгоритмі необхідно прийняти рішення, в якому порядку повинні перевірятись її значення. Для цього в деяких

випадках може виявитись ефективною евристика з **найменш обмежуваним значенням**. В ній надається перевага значенню, в якому з розгляду виключається найменша кількість варіантів вибору значень для сусідніх змінних у графі обмежень. Наприклад, припустимо, що на рис. 5.1 сформовано часткове присвоєння з $WA=red$ і $NT=green$ і що наступний варіант вибору призначений для Q . Синій колір був би поганим варіантом, оскільки виключив би останнє дозволене значення, що лишилось для сусідньої змінної відносно Q , тобто змінної SA . Тому евристика з «найменш обмежуваним значенням» надає перевагу значенню red , а не $blue$. Взагалі кажучи, в цій евристиці робиться спроба зберегти максимальну пружність для послідовних присвоєнь значень змінним. Безумовно, якби ми прагнули знайти всі розв'язки деякої задачі, а не перше з них, то таке впорядкування не мало б значення, оскільки нам так або інакше довелось би розглянути кожне значення. Таке ж зауваження залишається справедливим, якщо дана задача взагалі не має розв'язків.

2.12.4 Поширення інформації за допомогою обмежень

Досі в розглядуваному алгоритмі пошуку обмеження, що діють на якесь змінну, враховувались лише в той момент, коли відбувався відбір цієї змінної за допомогою функції Select-Unassigned-Variable. Але розглядаючи деякі обмеження на минулих етапах пошуку або навіть до початку пошуку, можна різко зменшити простір пошуку.

2.12.5 Попередня перевірка

Один зі способів кращого використання обмежень під час пошуку отримав назву попередньої перевірки (forward checking). При кожному присвоєнні значення змінній X в процесі попередньої перевірки проглядається кожна змінна Y з неприсвоєним значенням, котра поєднана з X за допомогою деякого обмеження, і з області визначення змінної Y видаляється будь-яке значення, яке є несумісним зі значенням, обраним для X . На рис. 5.4 показано хід пошуку розв'язку задачі розфарбування карти з допомогою попередньої перевірки. На основі даного прикладу можна зробити два важливих висновки. Перед усім варто відмітити, що після присвоєння $WA=red$ і $Q=green$ області

визначення змінних NT і SA зменшуються до єдиного значення; таким чином, розгалуження, пов'язане з пошуком значень для цих змінних, було повністю усунуто шляхом розповсюдження інформації, що стосується змінних WA і Q. Застосування евристики MRV, яка, безумовно, добре поєднується з попередньою перевіркою, дозволяє на наступному етапі автоматично обрати значення для змінних SA і NT. (Звісно, що попередня перевірка може розглядатись як ефективний спосіб інкрементного обчислення тієї інформації, яка потрібна евристиці MRV для виконання своєї роботи.) Другий висновок, що заслуговує на увагу, полягає у тому, що після присвоєння V=blue область визначення SA стає пустою. Тому попередня перевірка дозволила виявити, що часткове присвоєння {WA=red, Q=green, V=blue} є несумісним з обмеженнями цієї задачі, отже алгоритм одразу виконує повернення.

	WA	NT	Q	NSW	V	SA	T
Початкові області визначення	R G B	R G B	R G B	R G B	R G B	R G B	R G B
Після присвоєння WA=red	(R)	G B	R G B	R G B	R G B	G B	R G B
Після присвоєння Q=green	(R)	B	(G)	R B	R G B	B	R G B
Після присвоєння V=blue	(R)	B	(G)	R	(B)		R G B

Рис. 5.4. Пошук розв'язку задачі з розфарбуванням карти на основі попередньої перевірки. Спочатку виконується присвоєння WA=red, потім попередня перевірка приводить до видалення значень red з областей визначення сусідніх змінних NT і SA. Після присвоєння Q=green значення green видаляється з областей визначення NT, SA і NSW. Після присвоєння V=blue з областей визначення NSW і SA видаляється значення blue, в результаті чого змінна SA залишається без допустимих значень.

2.12.6 Поширення обмежень

Хоча попередня перевірка виявляє багато несумісностей, вона не дозволяє виявити їх всі. Наприклад, розглянемо третій рядок рис. 5.4, який показує, що якщо змінна WA має значення red, а Q – green, то обом змінним, NT і SA, має бути присвоєно значення blue. Але відповідні їм регіони є суміжними і тому не можуть мати одне й те саме значення кольору. Попередня перевірка не дозволяє виявити цю ситуацію як несумісність, оскільки не

передбачає достатньо далекий перегляд наперед. **Поширення обмежень** (constraint propagation) – це загальна назва методів поширення на інші змінні наслідків застосування деякого обмеження до однієї змінної; вданому випадку обхідно розповсюдити обмеження з WA і Q на NT і SA (як було зроблено за допомогою попередньої перевірки), а потім – на обмеження між NT і SA, аби виявити вказану несумісність. До того ж бажано, аби така операція виконувалася швидко: не має сенсу обмежувати таким чином об'єм пошуку, якщо буде витрачатись більше часу на поширення обмежень, ніж на виконання простого пошуку.

Ідея перевірки **сумісності дуг** лягла в основу швидкого методу поширення обмежень, який є значно більш потужним порівняно з попередньою перевіркою. В даному випадку термін дуга позначає орієнтоване ребро у графі обмежень, таке як дуга від SA до NSW. Якщо розглядаються поточні області визначення SA і NSW, то дуга є сумісною, якщо для кожного значення x змінної SA існує деяке значення у змінній NSW, яке сумісне з x . В третьому рядку на рис. 5.4 поточними областями визначення SA і NSW є {blue} і {red, blue} відповідно. При $SA=blue$ існує сумісне присвоєння для NSW, а саме $NSW=red$, тому дуга від SA до NSW сумісна. З іншого боку, зворотна дуга від NSW до SA несумісна, оскільки відносно присвоєння $NSW=blue$ не існує сумісного присвоєння для SA. Цю дугу можна зробити сумісною, видаливши значення blue з області визначення NSW.

На тому ж етапі в процесі пошуку перевірку сумісності дуг можна також застосувати до дуги від SA до NT. Третій рядок таблиці, наведеної на рис. 5.4, показує, що обидві змінні мають область визначення {blue}. Результатом стає те, що значення blue повинно бути видалено з області визначення SA, після чого ця область визначення залишається пустою. Таким чином, застосування перевірки сумісності дуг призвело до раннього виявлення тієї несумісності, яка не була виявлена за допомогою попередньої перевірки, що застосовується в чистому вигляді.

Перевірку сумісності дуг можна використовувати або в якості етапу попередньої обробки перед початком процесу пошуку, або в якості етапу поширення обмежень (аналогічно попередньої перевірки) після кожного присвоєння під час пошуку. (Останній алгоритм іноді називають MAC, скорочено позначає метод підтримки сумісності дуг – Maintaining Arc Consistency.) В обох випадках процес перевірки необхідно виконувати повторно, до тих пір, поки не перестануть виявлятись будь-які несумісності. Це пов'язано з тим, що при видаленні (задля усунення деякої несумісності дуг) будь-якого значення з області визначення деякої змінної може з'явитись нова несумісність дуг в тих дугах, які вказують на цю змінну. В повному алгоритмі перевірки сумісності дуг, AC-3, використовується черга для відстеження дуг, які повинні бути перевірені на несумісність (лістинг 5.2). Кожна дуга (x_i, x_j) по черзі «знімається з повістки дня» і перевіряється; якщо з області визначення x_i необхідно видалити які-небудь значення, то кожна дуга (x_k, x_i) , що вказує на x_i , повинна бути повторно додана до черги для перевірки. Складність перевірки сумісності дуг можна проаналізувати наступним чином: будь-яка бінарна задача CSP має не більше $O(n^2)$ дуг; кожна дуга (x_k, x_i) , може бути «внесена до повістки дня» лише d разів, оскільки область визначення x_i має не більше d значень, доступних для видалення; перевірка сумісності будь-якої дуги може виконана за час $O(d^2)$, тому в найгіршому випадку витрати часу складуть $O(n^2 d^3)$. Хоча такий метод є значно більш дорогим порівняно з попередньої перевіркою, всі ці додаткові витрати зазвичай виправдовуються.

Лістинг 5.2. Алгоритм перевірки сумісності дуг AC-3. Після використання алгоритму AC-3 кожна дуга або є сумісною, або деяка змінна має пусту область визначення, вказуючи на те, що цю задачу CSP неможливо зробити сумісною по лугам (і тому дана задача CSP не може бути вирішена). Позначення “AC-3” запропоновано розробником даного алгоритму [967], оскільки це – третя версія, що представлена в його статті.

```
function AC-3 (csp) returns визначення задачі CSP, можливо, зі зменшеними областями визначення змінних
```

inputs: csp, бінарна задача CSP зі змінними $\{X_1, X_2, \dots, X_n\}$

local variables: queue, черга, що складається з дуг, яка початково включає всі дуги з означення задачі csp

while черга queue не пуста **do**

$(x_i, x_j) \leftarrow Remove - First(queue)$

if Remove-Inconsistent-Values(x_i, x_j) **then**

for each X_k in Neighbors [X_i] **do**

додати (x_k, x_i) до черги queue

function Remove-Inconsistent-Values (x_i, x_j) **returns** значення true тоді і тільки тоді, коли відбулось видалення деякого значення

removed $\leftarrow false$

for each x in Domain [X_i] **do**

if жодне значення у з області визначення Domain [X_j] не дозволяє використовувати (x, y) для виконання обмеження між x_i та x_j

then видалити x з області визначення Domain [X_i];

removed $\leftarrow true$

return removed

Оскільки задачі CSP включають задачу 3SAT в якості часткового випадку, не варто розраховувати знайти алгоритми з поліноміальною часовою складністю, що дозволяють визначити, чи є дана конкретна задача CSP сумісною по дугам. Таким чином, можна зробити висновок, що метод перевірки сумісності дуг не дозволяє виявити всі можливі несумісності. Наприклад, як показано на рис. 5.1, часткове присвоєння {WA=red, NSW=red} несумісне, але алгоритм AC-3 не виявляє таку несумісність. Більш строгі форми поширення обмежень можна визначити за допомогою поняття **k-сумісності**. Задача CSP є k-сумісною, якщо для будь-якої множини k-1 змінних і для будь-

якого сумісного присвоєння значень цим змінним, будь якій k-й змінній завжди можна присвоїти деяке сумісне значення. Наприклад, 1-сумісність означає, що сумісною є кожна окрема змінна сама по собі; це поняття також називають **сумісністю вузла**. Далі, 2-сумісність – те ж, що і сумісність дуги, а 3-сумісність означає, що будь-яка пара суміжних змінних завжди може бути доповнена третьою сусідньою змінною; це поняття іменується також **сумісністю шляху**.

Будь-який граф називається **строго k-сумісним**, якщо він є k-сумісним, а також (k-1)-сумісним, (k-2)-сумісним, ... і т.д аж до 1-сумісного. Тепер припустимо, що є деяка задача CSP з вузлами n, яка зроблена строго n-сумісною (тобто строго k-сумісною для k=n). Тоді цю задачу можна розв'язати без повернень. Для цього спочатку можна обрати сумісне значення для X_1 . В такому випадку існує гарантія, що вдасться обрати значення для X_2 , оскільки граф є 2-сумісним, для X_3 , оскільки він – 3-сумісний, і т.д. Для кожної змінної X_i необхідно виконати пошук лише серед d значень в її області визначення, щоб знайти значення, сумісне з X_1, \dots, X_{i-1} . Це означає, що гарантується знаходження розв'язку за час $O(nd)$. Безумовно, за таку можливість також доводиться платити: будь який алгоритм забезпечення n-сумісності в найгіршому випадку повинен потребувати час, експоненціально залежний від n.

Між методами забезпечення n-сумісності і сумісності дуг існує цілий ряд проміжних методів, вибір яких здійснюється з врахуванням того, що для виконання більш строгих перевірок сумісності потрібно більше часу, але це дозволяє отримати більший ефект з точки зору скорочення коефіцієнту розгалуження і виявлення несумісних частинок присвоєнь. Існує можливість розрахувати таке найменше значення k, що виконання алгоритму перевірки k-сумісності буде гарантувати розв'язання даної задачі без повернень (див. розділ 5.4), але використання даного обчислення на практиці не завжди виправдано. Насправді визначення потрібного рівня перевірки сумісності в основному відноситься до області імперичних методів.

2.12.7 Обробка спеціальних обмежень

В реальних задачах часто зустрічаються деякі типи обмежень, які можуть оброблюватись за допомогою алгоритмів спеціального призначення, більш ефективних порівняно з методами загального призначення, які розглядались досі. Наприклад, обмеження Alldiff вказує, що всі змінні, що беруть в ньому участь, повинні мати різне значення (як в криптоарифметичній задачі). Одна з простих форм перевірки несумісності для обмежень Alldiff застосовується наступним чином: якщо в даному обмеженні беруть участь m змінних і всі вони разом взяті мають n можливих різних значень, при тому що $m > n$, то це обмеження не може бути виконано.

Використання даної перевірки призводить до створення наступного простого алгоритму: спочатку видалити з обмеження кожну змінну, що має одноelementну область визначення, потім видалити значення цієї змінної з областей визначення залишених змінних. Повторювати цю операцію до тих пір, поки є змінні з одноelementними областями визначення. Якщо в якийсь момент часу з'явиться пуста область визначення або залишиться більше змінних, ніж областей визначення, це буде означати, що виявлена несумісність.

Цей метод можна використати для виявлення несумісності в частковому присвоєнні $\{WA=\text{red}, NSW=\text{red}\}$ на рис. 5.1. Зверніть увагу на те, що змінні SA , NT і Q фактично пов'язані обмеженням Alldiff, оскільки кожна пара відповідних регіонів повинна бути позначена різними кольорами. Після застосування алгоритму AC-3 в поєднанні з цим частковим присвоєнням область визначенняожної змінної зменшується до $\{\text{green}, \text{blue}\}$. Це означає, що є три змінні і тільки два кольори, тому обмеження Alldiff порушується. Таким чином, іноді проста процедура перевірки сумісності для обмеження більш високого порядку ефективніше порівняно з процедурою перевірки сумісності дуг, що застосовується до еквівалентної множини бінарних обмежень.

Можливо, більш важливим обмеженням високого порядку є **ресурсне обмеження** (resource constraint), що іноді називають обмеженням «найбільше»

(atmost). Наприклад, припустимо, що PA_1, \dots, PA_4 позначають кількість персоналу, призначеного для виконання кожного з чотирьох завдань. Обмеження, згідно до якого може бути всього назначено не більше 10 членів персоналу, записується як $atmost(10, PA_1, PA_2, PA_3, PA_4)$. Несумісність можна виявити, перевіривши суму мінімальних значень поточних областей визначення; наприклад, якщо кожна змінна має область визначення $\{3, 4, 5, 6\}$, то обмеження $atmost$ не може бути виконано. Крім того, можна примусово досягти сумісності, видаляючи максимальне значення з будь-якої області визначення, якщо воно не сумісно з мінімальними значеннями інших областей визначення. Таким чином, якщо кожна змінна в даному прикладі має область визначення $\{2, 3, 4, 5, 6\}$, то з кожної області визначення можна видалити значення 5 і 6.

При розв'язанні великих задач перевірки ресурсних обмежень з цілочисленими значеннями (таких як задачі постачання, в яких передбачається переміщення тисяч людей у сотнях транспортних засобів) зазвичай не існує можливості представляти області визначення кожної змінної у вигляді великої множини цілих чисел і поступово скорочувати цю множину з використанням методів перевірки сумісності. Замість цього області визначення представляються у вигляді верхньої і нижньої границі і керуються за допомогою методу поширення цих границь. Наприклад, припустимо, що є два рейси, Flight271 і Flight272, в яких літаки мають відповідно місткість 165 і 385 пасажирів. Тому початкові області визначення для кількості пасажирів в кожному рейсі визначають наступним чином:

$$Flight271 \in [0, 165] \text{ і } Flight272 \in [0, 385]$$

Тепер припустимо, що є додаткове обмеження, згідно до якого в цих двох рейсах необхідно перевести 420 людей:

$$Flight271 + Flight272 \in [420, 420].$$

Поширюючи обмеження границь, можна зменшити області визначення до таких величин:

$$Flight271 \in [35, 165] \text{ і } Flight272 \in [255, 385]$$

Задача CSP називається сумісною з границями (bounds-consistent), якщо для кожної змінної X , а також одночасно для нижнього і верхнього граничних значень X існує деяке значення Y , яке виконує задає обмеження між X та Y для кожної змінної Y . Такого роду **поширення границь** (bounds propagation) широко використовується в практичних задачах з обмеженнями.

2.12.8 Інтелектуальний пошук з поверненнями: пошук в зворотному напрямку

В алгоритмі Backtracking-Search, наведеному в лістингу 5.1, застосувалось дуже просте правило, про те, що робити, якщо якась гілка пошуку закінчується невдачею: повернутись до минулоЯ змінної і спробувати для неї інше значення. Такий метод називається **хронологічним пошуком з поверненнями**, оскільки повторно відвідується пункт, в якому було прийняте останнє по часу рішення. В даному підрозділі буде показано, що існують кращі способи пошуку з поверненнями.

Розглянемо, що станеться в випадку застосування простого пошуку з поверненнями в задачі, показаній на рис. 5.1, з постійним впорядкуванням змінних Q , NSW, V, T, SA, WA, NT. Припустимо, що сформовано часткове присвоєння $\{Q=\text{red}, \text{NSW}=\text{green}, \text{V}=\text{blue}, \text{T}=\text{red}\}$. Після спроби присвоїти значення наступній змінній, SA, буде виявлено, що будь-яке значення порушує якесь обмеження. Алгоритм повертається до вузла T і намагається назначити новий колір для Тасманії! Очевидно, що це безглуздо, оскільки зміна кольору Тасманії не дозволяє розв'язати проблему Південної Австралії.

Більш інтелектуальних підхід до пошуку з поверненнями полягає у тому, аби повернутись до однієї з множин змінних, які стали причиною невдачі. Ця множина називається **конфліктною множиною**; в даному випадку конфліктною множиною для SA є $\{Q, \text{NSW}, \text{V}\}$. Взагалі кажучи, конфліктна множина для змінної X являє собою множину змінних з раніше присвоєнimi значеннями, які пов'язані з X обмеженнями. Метод **зворотного переходу** виконує зворотний перехід до змінної з останнім по часу присвоєним значенням з конфліктної множини; в даному випадку в зворотного переході

варто перескочити через вузол Тасманії і спробувати застосувати нове значення для V. Така операція може бути легко реалізована шляхом модифікації алгоритму Backtracking-Search таким чином, щоб він накопичував дані про конфліктну множину, одночасно перевіряючи одне зі значень, дозволених для присвоєння. Якщо не буде знайдено жодного дозволеного значення, алгоритм повинен повернутись до останнього по часу елементу конфліктної множини і наряду з ним встановити індикатор невдачі.

Уважний читач вже повинен був помітити, що попередня перевірка дозволяє визначити конфліктну множину без додаткової роботи, оскільки кожен раз, коли процедура попередньої перевірки, що базується на присвоєнні значення змінній X, видає деяке значення з області визначення Y, вона повинна додати X до конфліктної множини Y. Крім того, кожен раз, коли це останнє значення видається з області визначення Y, змінні з конфліктної множини Y добавляються до конфліктної множини X. В цьому випадку після переходу до Y можна одразу ж встановити, куди повинен бути виконаний зворотний перехід, якщо це необхідно.

А проникливий читач вже повинен був помітити дещо дивне: зворотний перехід відбувається, коли кожне значення в деякій області визначення конфліктує з поточним присвоєнням, але попередня перевірка виявляє цей випадок і взагалі виключає можливість досягнення такого вузла! Насправді можна показати, що кожна гілка, відсічена за допомогою зворотного переходу, відсікається також попередньою перевіркою. Тому простий пошук зі зворотним переходом в поєднанні з пошуком з попередньою перевіркою стає надлишковим, а фактично обраний перехід надлишковий в будь-якому пошуку, в якому використовується більш строга перевірка сумісності, така як МАС.

Незважаючи на зауваження, зроблені в попередньому абзаці, ідея, що лежить в основі зворотного переходу, залишається перспективною, якщо повернення здійснюється з урахуванням причин невдачі. При зворотному переході невдача виявляється після того, як область визначення деякої змінної стає порожньою, але в багатьох випадках якесь гілка пошуку стає

безперспективною задовго до того, як це відбувається. Ще раз розглянемо часткове присвоювання $\{WA = \text{red}, NSW = \text{red}\}$ (яке відповідно до наведеного вище опису є несумісним). Припустимо, що на наступному етапі робиться спроба привласнення $T = \text{red}$, а потім здійснюється присвоювання значень змінним NT, Q, V, SA . Відомо, що жодне присвоювання не застосовується для цих останніх чотирьох змінних, тому в кінцевому результаті не залишається доступних значень, які можна було б спробувати привласнити змінній NT . Тепер виникає питання, куди виконати повернення? Зворотний перехід не може застосовуватися, оскільки в області визначення змінної NT є значення, сумісні зі значеннями, що були раніше присвоєні змінним, - змінна NT не має повної конфліктної множини змінних з раніше присвоєнними значеннями, яке викликало б невдачу при спробі привласнення її значення. Однак відомо, що невдачу викликають чотири змінні, NT, Q, V і SA , разом узяті, оскільки в множині змінних з раніше присвоєнними значеннями повинні існувати такі змінні, які безпосередньо конфліктують з цими чотирма. Ці міркування приводять до створення більш глибокого визначення поняття конфліктної множини для такої змінної, як NT : конфліктною множиною називається множина змінних з раніше присвоєнними значеннями, які, наряду з усіма змінними зі значеннями, що привласнюються в подальшому, стають причиною того, що для NT не існує сумісного рішення. В такому випадку конфліктна множина складається з змінних WA і NSW , тому алгоритм повинен виконати повернення до NSW і пропустити змінну, що відповідає Тасманії. Алгоритм зворотного переходу, в якому використовуються конфліктні множини, визначений таким чином, називається **алгоритмом зворотного переходу, керованого конфліктами** (conflict-directed backjumping).

Тепер необхідно пояснити, як обчисляються ці нові конфліктні множини. Метод, що застосовується для цього, фактично дуже простий. «Кінцева» невдача в якісь гілці пошуку завжди виникає через те, що область визначення деякої змінної стає пустою; ця змінна має типову конфліктну множину. В даному прикладі невдача виникає при присвоєнні значення змінній SA , а її конфліктною множиною є (скажімо) $\{WA, NT, Q\}$. Виконується зворотний

перехід до змінної Q , і ця змінна поглинає конфліктну множину, отриману від SA (безумовно, після виключення із нього самої змінної Q), поєднуючи його зі своєю власною прямую конфліктною множиною, яка є $\{NT, NSW\}$; новою конфліктною множиною стає $\{WA, NT, NSW\}$. Це означає, що, продовжуючи пошук від Q , не можна знайти розв'язок при наявності раніше виконаного присвоєння змінним $\{WA, NT, NSW\}$. Тому зворотний перехід виконується до змінної NT , присвоєння значення якій виконано останнім по часу порівняно з іншими змінними. Змінна NT поглинає множину $\{WA, NT, NSW\} - \{NT\}$, поєднуючи її зі своєю власною прямую конфліктною множиною $\{WA\}$, що призводить до отримання $\{WA, NSW\}$ (як було вказано в минулому абзаці). Тепер алгоритм виконує зворотний перехід до NSW , як і передбачалось. Підведемо підсумки: припустимо, що X_j – поточна змінна, а $conf(X_i)$ – її конфліктна множина. Якщо всі спроби присвоєння для кожного можливого значення змінної X_j закінчуються невдачею, то необхідно виконати повернення до змінної X_i з останнім по часу присвоєним значенням в множині $conf(X_j)$ і встановити:

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}$$

Зворотний перехід, керований конфліктами, дозволяє повернутись в правильну точку дерева пошуку, але не запобігає можливості появи таких же помилок в іншій гілці дерева. Метод **визначення обмежень за допомогою навчання** (constraint learning) по суті представляє собою метод модифікації задачі CSP шляхом додавання нового обмеження, висунутого на основі логічного аналізу цих конфліктів.

3 ТЕМА З СКЛАДНІ СТРУКТУРИ ДАНИХ

Структура даних (*data structure*) – програмна одиниця, що дозволяє зберігати і обробляти безліч однотипних і/або логічно пов'язаних даних в обчислювальній техніці. Для додавання, пошуку, зміни і видалення даних структура даних надає певний набір функцій, з яких складається інтерфейс.

Термін «структурі даних» може мати кілька близьких, але тим не менше різних значень: випрвити

- абстрактний тип даних;
- реалізація будь-якого абстрактного типу даних;
- екземпляр типу даних, наприклад, конкретний список;
- в контексті функціонального програмування - унікальна одиниця,

що зберігається при змінах. Про неї неформально говорять як про одну структуру даних, незважаючи на можливу наявність різних версій.

Різні види структур даних підходять для різних додатків; деякі з них мають вузьку спеціалізацію для певних завдань. Наприклад, В-дерева зазвичай підходять для створення баз даних, в той час як хеш-таблиці використовуються повсюдно для створення різного роду словників, наприклад, для відображення доменних імен в інтернет-адреси комп'ютерів.

3.1 Індексні файли

Незважаючи на високу ефективність хеш-адресації, в файлових структурах далеко не завжди вдається знайти відповідну функцію, тому при організації доступу по первинному ключу широко використовуються індексні файли. У деяких комерційних системах індексними файлами називаються також і файли, організовані у вигляді інвертованих списків, які використовуються для доступу по вторинному ключу. Ми будемо дотримуватися класичної інтерпретації індексних файлів і сподіваємося, що якщо ви зіткнетесь з іншою інтерпретацією, то зумієте розібратися в суті, не дивлячись на деяку плутанину в термінології. Напевно, це частково пов'язано з тим, що область баз даних є досить молодий галуззю знань, і не дивлячись на те, що тут уже виробилася певна термінологія, багато постачальників

комерційних СУБД надають перевагу своєму спрощеному сленгу при описі власних продуктів. Іноді це пов'язано з тим, що з метою реклами вони не хочуть посилатися на старі, добре відомі моделі і методи організації інформації в системі, а винаходять нові назви при описі своїх моделей, тим самим намагаючись розрекламувати ефективність своїх продуктів. Добре знання принципів організації даних допоможе вам об'єктивно оцінювати рішення, пропоновані постачальниками сучасних СУБД, і не потрапляти на рекламні гачки.

Індексні файли можна уявити як файли, що складаються з двох частин. Це не обов'язково фізичне поєднання цих двох частин в одному файлі, в більшості випадків індексна область утворює окремий індексний файл, а основна область утворює файл, для якого створюється індекс. Але нам зручніше розглядати ці дві частини спільно, так як саме взаємодія цих частин і визначає використання механізму індексації для прискорення доступу до записів.

Ми припускаємо, що спочатку йде індексна область, яка займає деякий ціле число блоків, а потім йде основна область, в якій послідовно розташовані всі записи файлу.

Залежно від організації індексної і основної областей розрізняють 2 типу файлів: з *щільним індексом* і з *нешільним індексом*. Ці файли мають ще додаткові назви, які безпосередньо пов'язані з методами доступу до довільного запису, які підтримуються даними файловими структурами.

Файли з щільним індексом називаються також індексного-прямими файлами, а файли з нещільним індексом називаються також індексного-послідовними файлами. Сенс цих назв нам буде ясний після того, як ми більш детально розглянемо механізми організації даних файлів.

3.1.1 Файли з щільним індексом, або індексного-прямі файли

Розглянемо файли з *щільним індексом*. У цих файлах основна область містить послідовність записів однакової довжини, розташованих в довільному порядку, а структура індексного запису в них має наступний вигляд:

значення ключа	Номер запису в основному файлі
----------------	--------------------------------

Тут *значення ключа* – це значення первинного ключа, а номер запису – це порядковий номер запису в основний області, яка має дане значення первинного ключа.

Так як індексні файли будуються для первинних ключів, які однозначно визначають запис, у них не може бути двох записів, що мають одинакові значення первинного ключа. В індексних файлах з щільним індексом для кожного запису в основний області існує одна запис з індексного області. Всі записи в індексній області впорядковані за значенням ключа, тому можна застосувати більш ефективні способи пошуку в упорядкованому просторі.

Довжина доступу до довільного запису оцінюється не в абсолютних значеннях, а в кількості звернень до пристрою зовнішньої пам'яті, яким зазвичай є диск. Саме звернення до диска є найбільш тривалою операцією в порівнянні з усіма обробками в оперативній пам'яті.

Найбільш ефективним алгоритмом пошуку на впорядкованому масиві є логарифмічний, або бінарний, пошук. Максимальна кількість кроків пошуку визначається двійковим логарифмом від загального числа елементів в шуканому просторі пошуку:

$$T_n = \log_2 N, \text{ де } N — \text{ число елементів.}$$

Однак в нашему випадку є суттєвим тільки число звернень до диска при пошуку запису по заданому значенню первинного ключа. Пошук відбувається в індексній області, де застосовується двійковий алгоритм пошуку індексного запису, а потім шляхом прямої адресації ми звертаємося до основної області вже по конкретному номеру запису. Для того щоб оцінити максимальний час

доступу, нам треба визначити кількість звернень до диска для пошуку довільного запису.

На диску записи файлів зберігаються в блоках. Розмір блоку визначається фізичними особливостями дискового контролера і операційною системою. В одному блоці можуть розміщуватися декілька записів. Тому нам треба визначити кількість індексних блоків, який буде потрібно для розміщення всіх необхідних індексних записів, а тому максимальне число звернень до диска дорівнюватиме двійковому логарифму від заданого числа блоків плюс одиниця. Навіщо потрібна одиниця? Після пошуку номера запису в індексному області ми повинні ще звернутися до основної області файла. Тому формула для обчислення максимального часу доступу в кількості звернень до диска виглядає наступним чином:

$$T_n = \log_2 N_{\text{бл.інд.}} + 1.$$

Давайте розглянемо конкретний приклад і порівняємо час доступу при послідовному перегляді і при організації щільного індексу.

Припустимо, що ми маємо такі вихідні дані:

Довжина запису файла (LZ) - 128 байт. Довжина первинного ключа (LK) - 12 байт. Кількість записів в файлі (KZ) - 100000. Розмір блоку (LB) - 1024 байт.

Розрахуємо розмір індексного запису. Для представлення цілого числа в межах 100000 нам буде потрібно 3 байта, можемо вважати, що у нас допустима тільки парна адресація, тому нам треба відвести 4 байта для зберігання номера запису, тоді довжина індексного запису буде дорівнює сумі розміру ключа і посилання на номер запису, то є:

$$LI = LK + 4 = 12 + 4 = 16 \text{ байт.}$$

Визначимо кількість індексних блоків, які потрібні для забезпечення посилань на задану кількість записів. Для цього спочатку визначимо, скільки індексних записів може зберігатися в одному блоці:

$$KIZB = LB/LI = 1024/16 = 64 \text{ індексних записів в одному блоці.}$$

Тепер визначемо необхідну кількість індексних блоків:

$$KIB = KZ/KZIB = 100000/64 = 1563 \text{ блока.}$$

Ми округлили в більшу сторону, тому що простір виділяється цілими блоками, і останній блок у нас буде заповнений не повністю.

А тепер ми вже можемо обчислити максимальну кількість звернень до диска при пошуку довільного запису:

$$T_{\text{пошуку}} = \log_2 KIB + 1 = \log_2 1563 + 1 = 11 + 1 = 12 \text{ звернень до диска.}$$

Логарифм ми теж округляємо, так як вважаємо кількість звернень, а воно повинно бути цілим числом.

Отже, для пошуку довільного запису по первинному ключу при організації щільного індексу буде потрібно не більше 12 звернень до диска. А тепер оцінимо, який виграш ми отримуємо, адже організація індексу пов'язана з додатковими накладними витратами на його підтримку, тому така організація може бути виправдана тільки в тому випадку, коли вона дійсно дає значний виграш. Якби ми не створювали індексний простір, то при довільному зберіганні записів в основний області нам би в гіршому випадку було необхідно переглянути всі блоки, в яких зберігається файл, часом перегляду записів всередині блоку ми нехтуємо, так як цей процес відбувається в оперативній пам'яті.

Кількість блоків, яка необхідна для зберігання всіх 100 000 записів, ми визначимо за такою формулою:

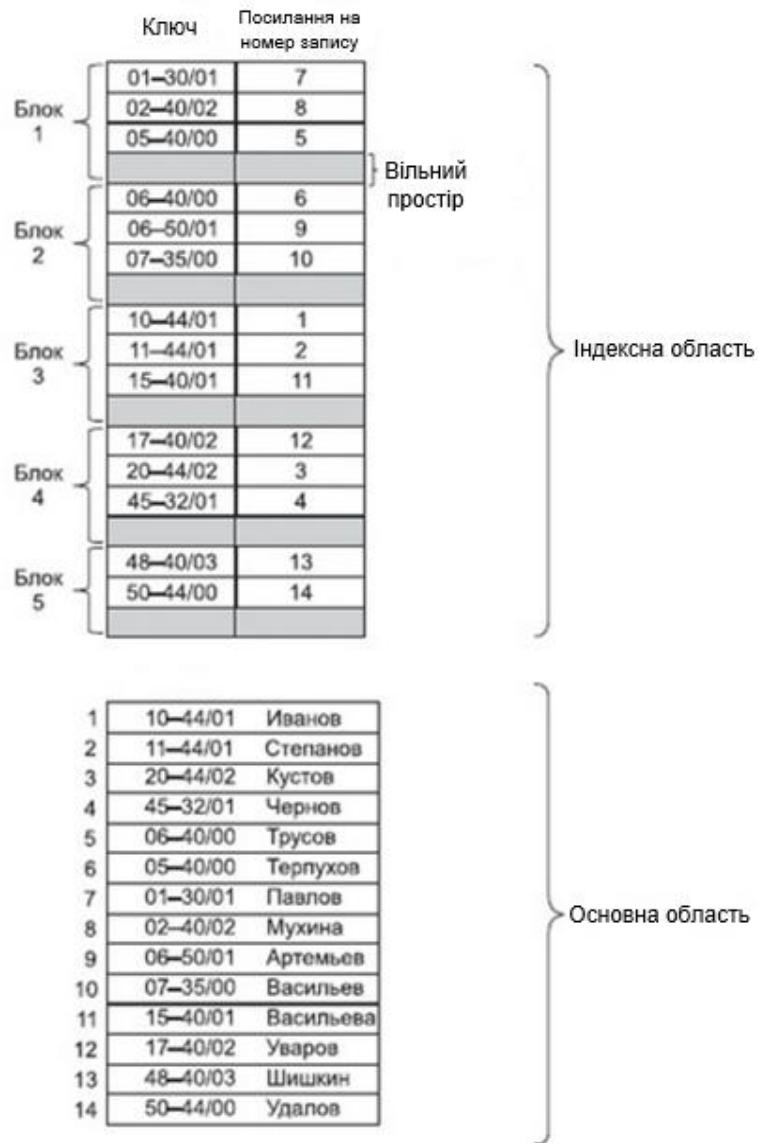
$$KBO = KZ/(LB/LZ) = 100000/(1024/128) = 12500 \text{ блоків.}$$

І це означає, що максимальний час доступу дорівнює 12500 звернень до диска. Так, дійсно, виграш істотний.

Розглянемо, як здійснюються операції додавання і видалення нових записів.

При операції додавання здійснюється запис в кінець основної області. В індексній області необхідно провести занесення інформації в конкретне місце,

щоб не порушувати впорядкованості. Тому вся індексна область файлу розбивається на блоки і при початковому заповненні в кожному блоці залишається вільна область (відсоток розширення).



Після визначення блоку, в який повинен бути занесений індекс, цей блок копіюється в оперативну пам'ять, там він модифікується шляхом вставки в потрібне місце нового запису (благо в оперативній пам'яті це робиться на кілька порядків швидше, ніж на диску) і, змінений, записується назад на диск.

Визначимо максимальну кількість звернень до диска, яке потрібно при додаванні запису, - це кількість звернень, необхідне для пошуку запису плюс одне звернення для занесення зміненого індексного блоку і плюс одне звернення для занесення запису в основну область.

$$T_{\text{додавання}} = \log_2 N + 1 + 1 + 1 .$$

Природно, в процесі додавання нових записів відсоток розширення постійно зменшується. Коли зникає вільна область, виникає переповнення індексної області. У цьому випадку можливі два рішення: або перебудувати заново індексну область, або організувати область переповнення для індексної області, в якій будуть зберігатися записи, що не помістилися в основну область. Однак перший спосіб потребує додаткового часу на перебудову індексної області, а другий збільшить час на доступ до довільного запису і потребує організації додаткових посилань в блоках на область переповнення.

Саме тому при проектуванні фізичної бази даних так важливо заздалегідь якомога точніше визначити обсяги інформації, що зберігається, спрогнозувати її зростання і передбачити відповідне розширення області зберігання.

При видаленні запису виникає наступна послідовність дій: запис в основний області позначається як відсутня, в індексному області відповідний індекс знищується фізично, тобто записи, які йдуть за видаленим записом, переміщаються на її місце і блок, в якому зберігався даний індекс, заново записується на диск. При цьому кількість звернень до диска для цієї операції таке ж, як і при додаванні нового запису.

3.1.2 Файли с нещільним індексом, или індексно-послідовні файли

Спробуємо вдосконалити спосіб зберігання файлу: будемо зберігати його в упорядкованому вигляді і застосуємо алгоритм двійкового пошуку для доступу до довільного запису. Тоді час доступу до довільного запису буде істотно менше. Для нашого прикладу це буде:

$$T = \log_2 KBO = \log_2 12500 = 14 \text{ звернень до диска.}$$

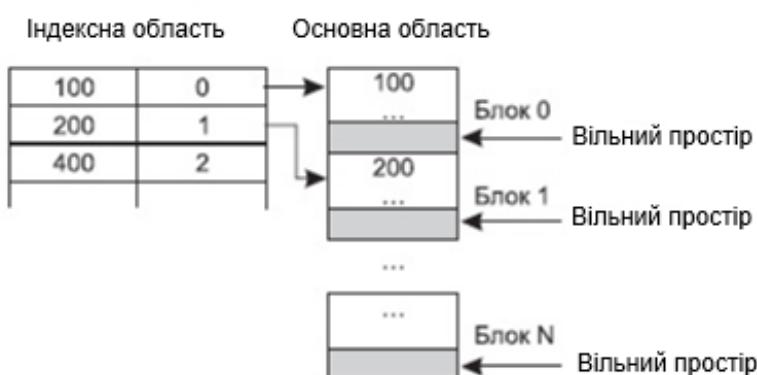
І це суттєво менше, ніж 12 500 звернень при довільному зберіганні записів файлу. Однак і підтримання основного файла в упорядкованому вигляді також операція складна.

Нещільний індекс будеться саме для упорядкованих файлів. Для цих файлів використовується принцип внутрішнього упорядкування для зменшення

кількості збережених індексів. Структура запису індексу для таких файлів має такий вигляд:

Значення ключа першого запису блоку	Номер блоку із цим записом
-------------------------------------	----------------------------

В індексній області ми тепер шукаємо потрібний блок за заданим значенням первинного ключа. Так як всі записи впорядковані, то значення першого запису блоку дозволяє нам швидко визначити, в якому блоці знаходиться шуканий запис. Всі інші дії відбуваються в основній області. На малюнку представлений приклад заповнення основної та індексної областей, якщо первинним ключем є цілі числа.



Час сортування великих файлів досить значний, але оскільки файли підтримуються відсортованими з моменту їх створення, накладні витрати в процесі додавання нової інформації будуть набагато менше.

Оцінимо час доступу до довільного запису для файлів з нещільним індексом. Алгоритм розв'язання задачі аналогічний.

Спочатку визначимо розмір індексного запису. Якщо раніше посилання розраховувалася виходячи з того, що було потрібно посилатися на 100 000 записів, то тепер нам потрібно посилатися лише на 12 500 блоків, тому для посилання досить двох байт. Тоді довжина індексного запису буде дорівнює:

$$LI = LK + 2 = 14 + 2 = 14 \text{ байт.}$$

Тоді кількість індексних записів в одному блоці дорівнюватиме:

$$KIZB = LB/LI = 1024/14 = 73 \text{ індексних записів в одному блоці.}$$

Визначимо кількість індексних блоків, потрібних для зберігання необхідних індексних записів:

$$KIB = KBO/KZIB = 12500/73 = 172 \text{ блока.}$$

Тоді час доступу за минулою формулою буде визначатися:

$$T_{\text{поиска}} = \log_2 KIB + 1 = \log_2 172 + 1 = 8 + 1 = 9 \text{ звернень до диска.}$$

Ми бачимо, що при переході до нещільного індексу час доступу зменшився практично в півтора рази. Тому можна визнати, що організація нещільного індексу дає виграш в швидкості доступу.

Розглянемо процедури додавання та видалення нового запису при подібному індексі.

Тут механізм включення нового запису принципово різний від раніше розглянутого. Тут новий запис повинен заноситися відразу в необхідний блок на необхідне місце, яке визначається заданим принципом впорядкованості на безлічі значень первинного ключа. Тому спочатку шукається необхідний блок основної пам'яті, в який треба помістити новий запис, а потім цей блок читається, потім в оперативній пам'яті коригується вміст блоку і він знову записується на диск на старе місце. Тут, так само як і в першому випадку, повинен бути заданий відсоток первинного заповнення блоків, але тільки стосовно основної області. В MS SQL server цей відсоток називається Full-factor і використовується при формуванні кластеризованих індексів. Кластеризованими називаються якраз індекси, в яких вихідні записи фізично впорядковані за значеннями первинного ключа. При внесенні нового запису індексна область не коригується.

Кількість звернень до диска при додаванні нового запису дорівнює кількості звернень, необхідних для пошуку відповідного блоку плюс одне звернення, яке потрібно для занесення зміненого блоку на старе місце.

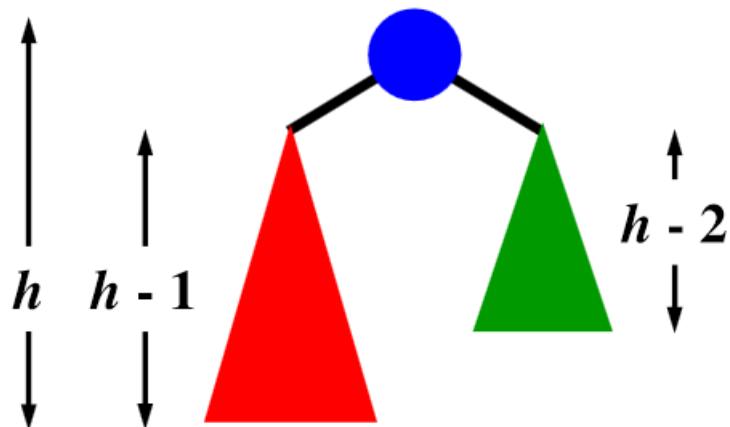
$$T_{\text{додавання}} = \log_2 N + 1 + 1 \text{ звернень.}$$

Знищення запису відбувається шляхом її фізичного видалення з основної області, при цьому індексна область зазвичай не коригується, навіть якщо

видаляється перший запис блоку. Тому кількість звернень до диска при видаленні запису таке ж, як і при додаванні нового запису.

3.2 АВЛ-дерево

АВЛ-дерево - збалансоване по висоті двійкове дерево пошуку: для кожної його вершини висота її двох піддерев відрізняється не більше ніж на 1.



АВЛ - абревіатура, утворена першими літерами прізвищ авторів (радянських вчених) Георгія Максимовича Адельсон-Бельського і Євгена Михайловича Ландіса.

Максимальна висота АВЛ-дерева при заданому числі вузлів:

$$h \leq \lfloor 1.45 \log(n + 2) \rfloor$$

Основна ідея

Якщо вставка або видалення елемента призводить до порушення збалансованості дерева, то виконується його балансування.

Коефіцієнт збалансованості вузла (balance factor) - це різниця висот його лівого і правого піддерев.

У АВЛ-дереві коефіцієнт збалансованості будь-якого вузла приймає значення з множини $\{-1, 0, 1\}$.

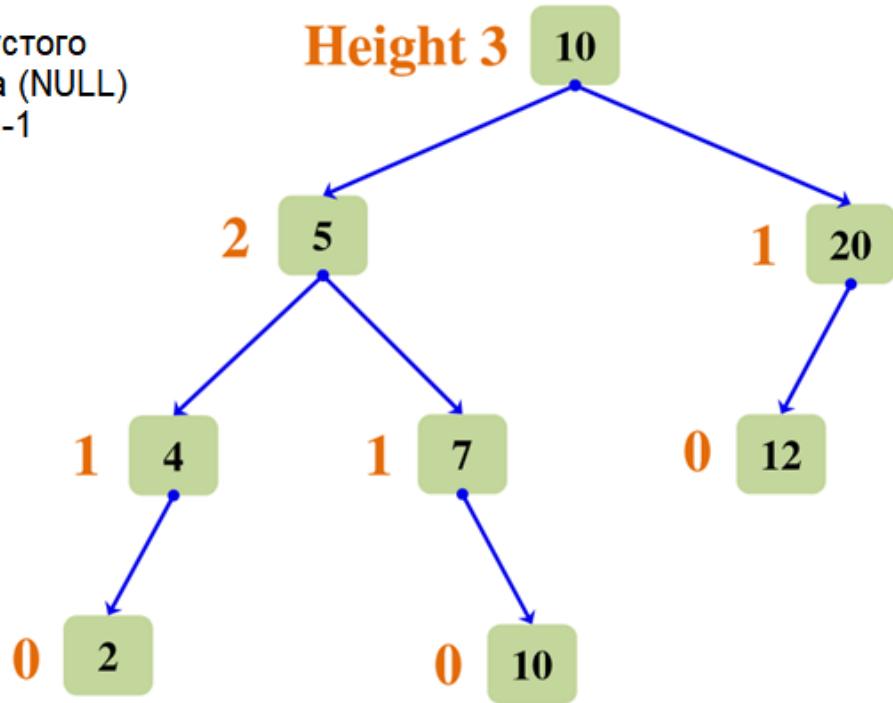
Висота вузла (height) - це довжина найбільшого шляху від нього до дочірнього вузла, що є листом.

Висота листа дорівнює 0.

Висота порожнього піддерева (NULL) дорівнює -1.

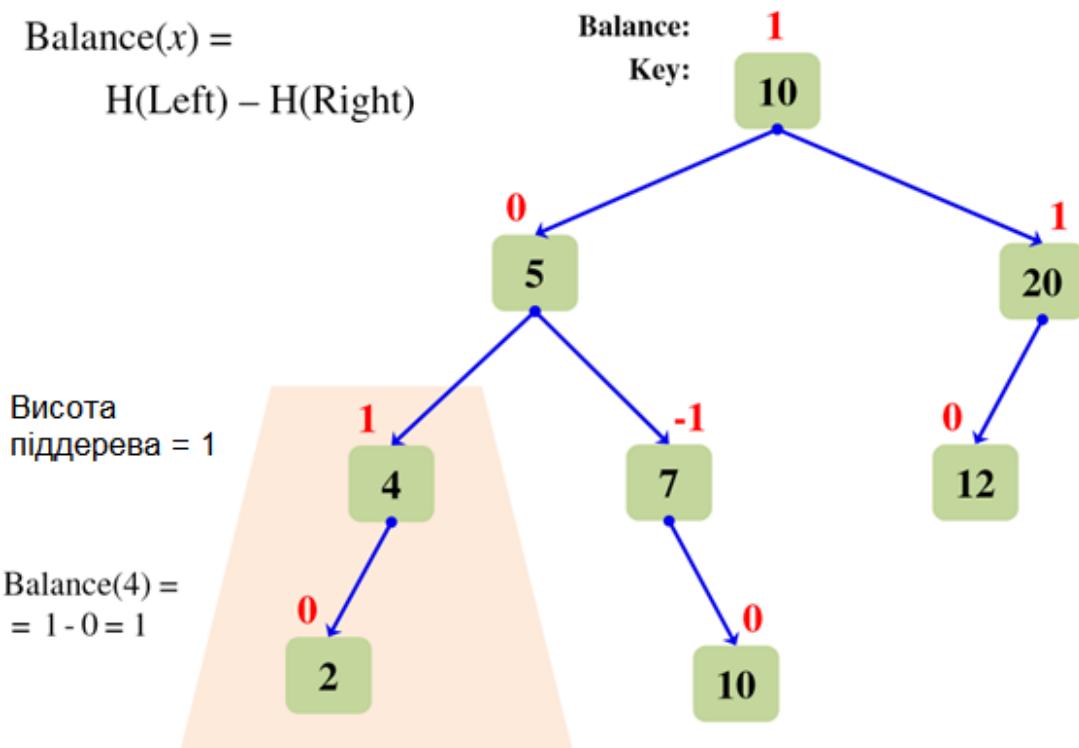
Висота вузла (Node height) – H.

Висота пустого
піддерева (NULL)
дорівнює -1



Коефіцієнт збалансованості

$\text{Balance}(x) =$
 $H(\text{Left}) - H(\text{Right})$



Випадок порушення балансу:

$\text{Balance}(x) =$
 $H(\text{Left}) - H(\text{Right})$

Не AVL-дерево

Висота піддерева = 1

Balance(4) = 1 - 0 = 1

Balance:
Key:

2
10

0
20

0
5

1
4

-1
7

0
10

0
2

Балансування дерева (Rebalancing)

Після додавання нового елемента необхідно оновити коефіцієнти збалансованості батьківських вузлів.

Якщо будь-який батьківський вузол прийняв значення -2 або 2, то необхідно виконати балансування піддерева шляхом повороту (rotation).

Типи поворотів:

Одиночний правий поворот (R-rotation, single right rotation).

Виконується після додавання елемента в ліве піддерево лівого дочірнього вузла дерева.

Одиночний лівий поворот (L-rotation, single left rotation).

Виконується після додавання елемента в праве піддерево правого дочірнього вузла дерева.

Подвійний ліво-правий поворот (LR-rotation, double left-right rotation).

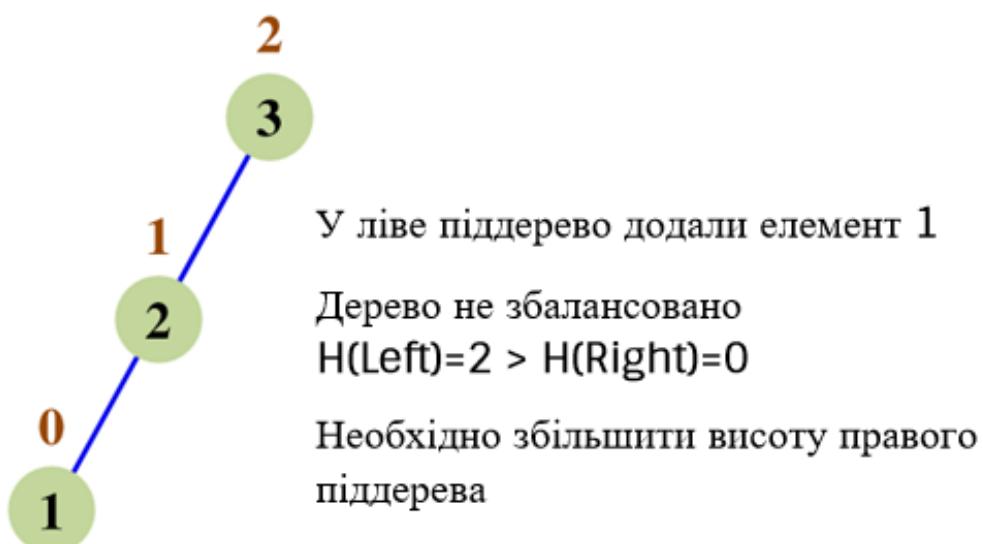
Виконується після додавання елемента в праве піддерево лівого дочірнього вузла дерева.

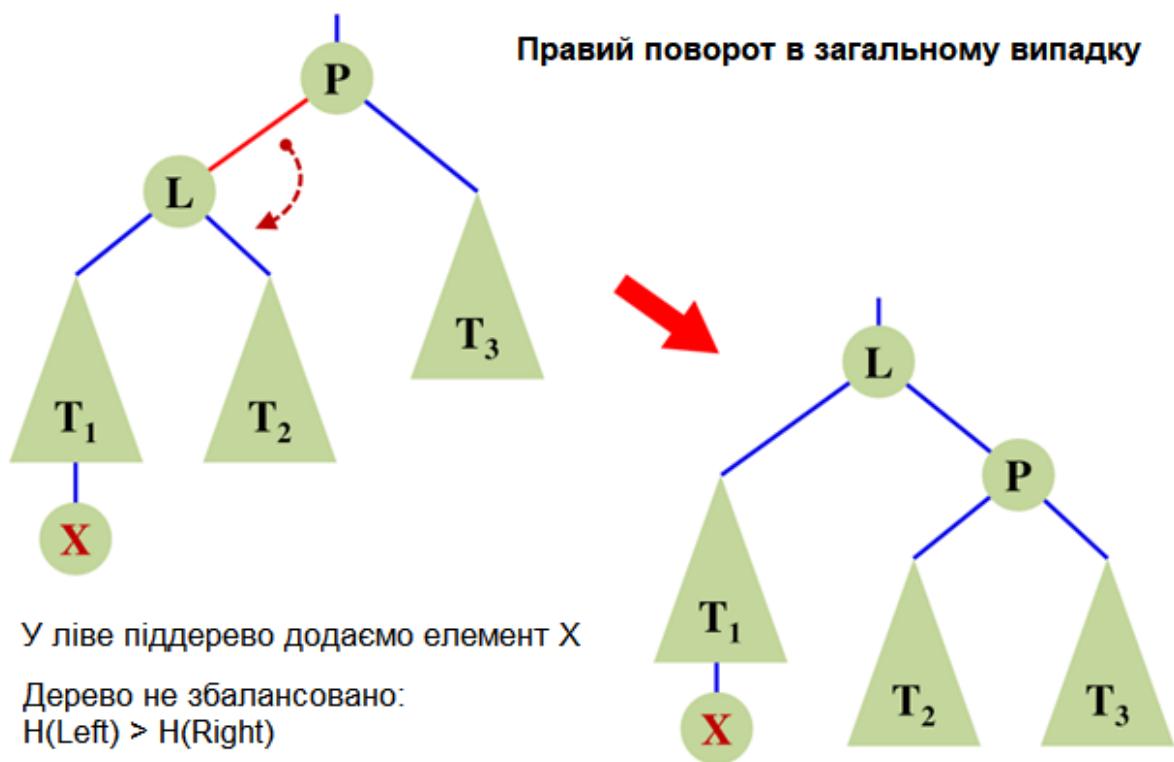
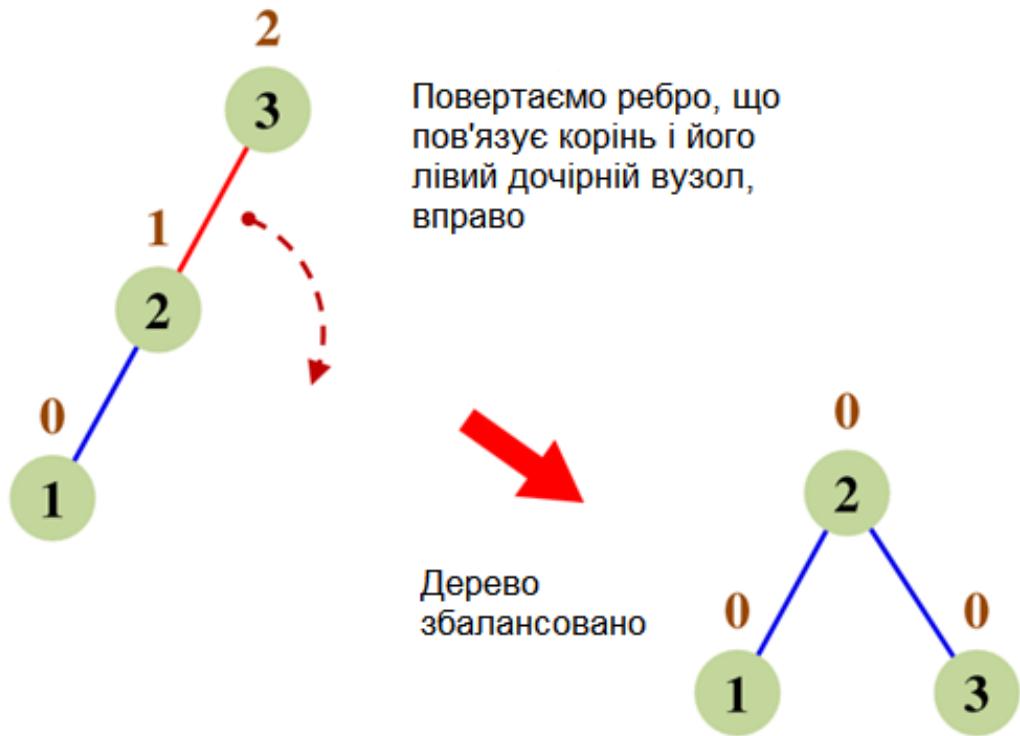
Подвійний право-лівий поворот (RL-rotation, double right-left rotation).

Виконується після додавання елемента в ліве піддерево правого дочірнього вузла дерева.

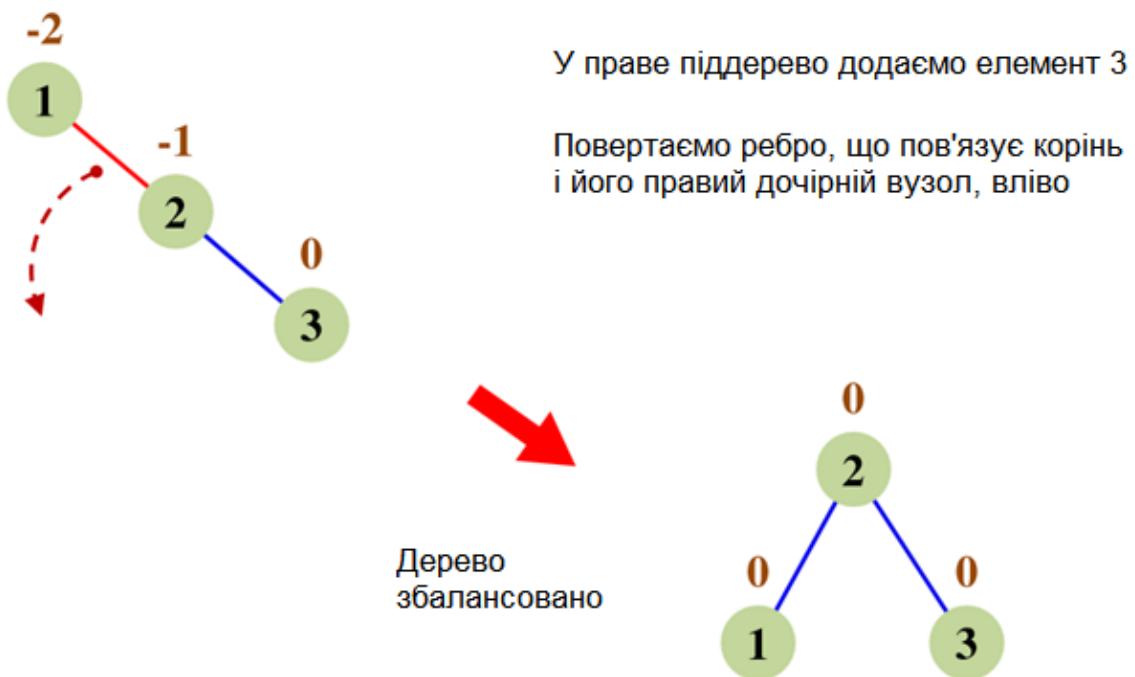
Розглянемо деякі повороти детальніше.

Одиночний правий поворот (R-rotation, single right rotation).



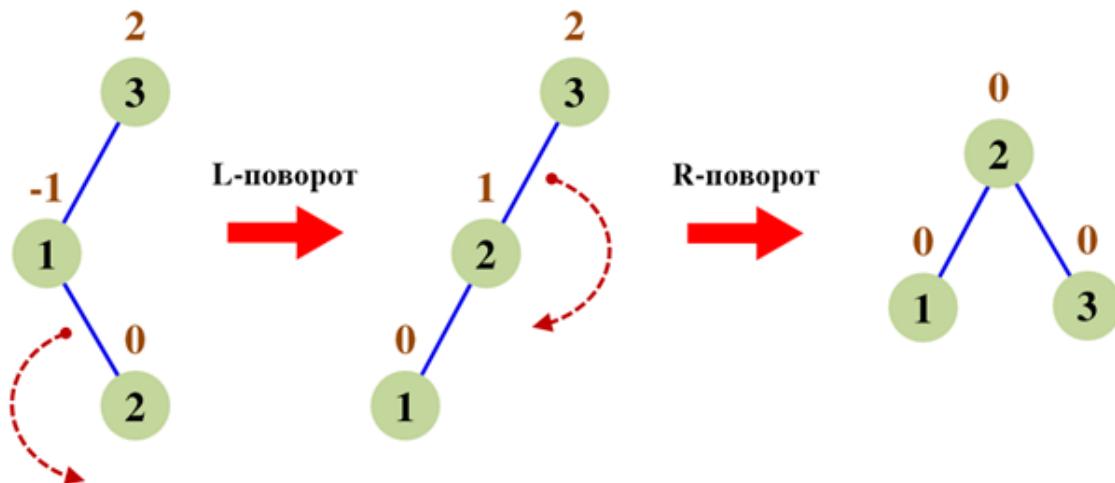


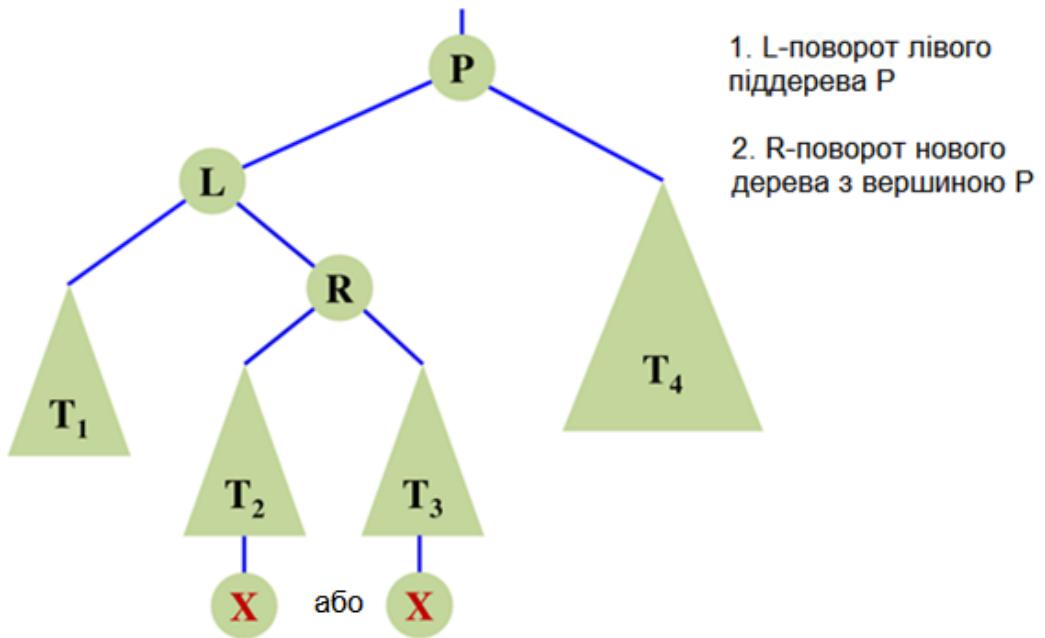
Одиночний лівий поворот (L-rotation, single left rotation)



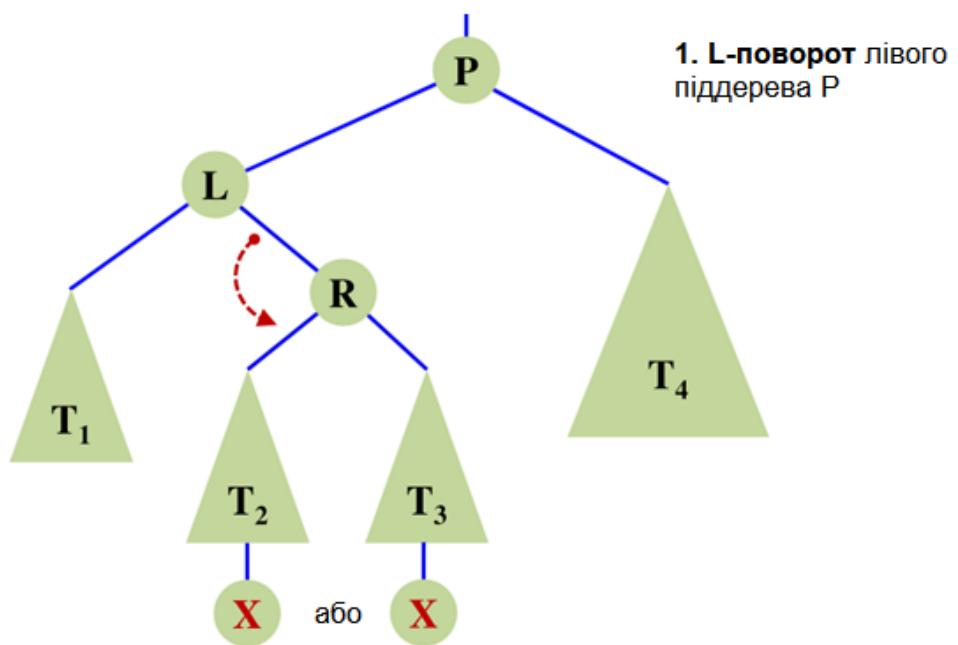
Подвійний ліво-правий поворот (LR-rotation, double left-right rotation)

LR-поворот виконується після додавання елемента в праве піддерево лівого дочірнього вузла

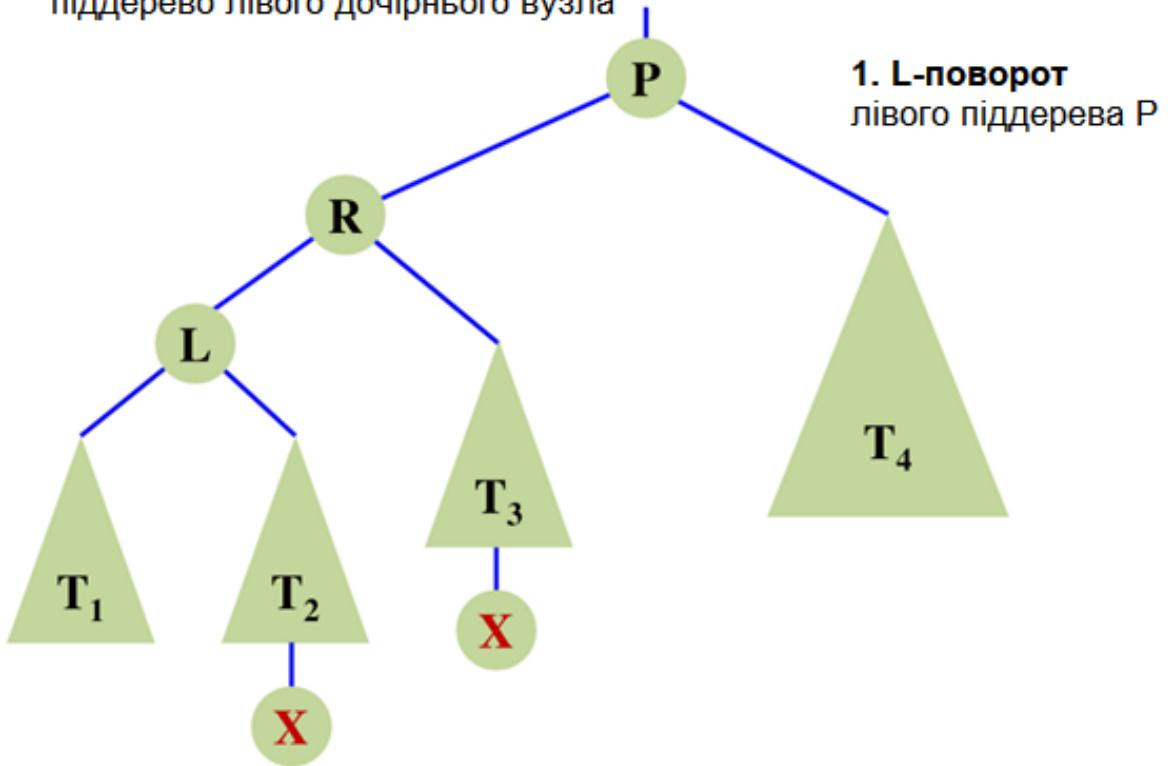




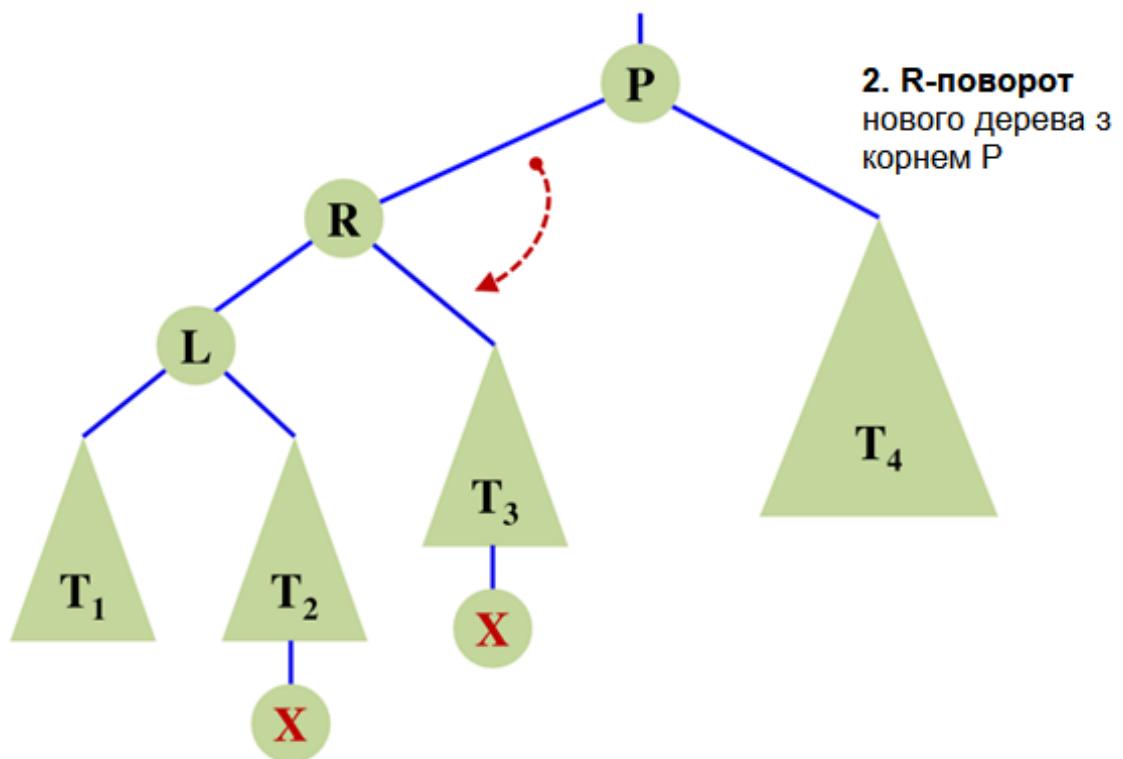
LR-поворот виконується після додавання елемента в праве піддерева лівого дочірнього вузла



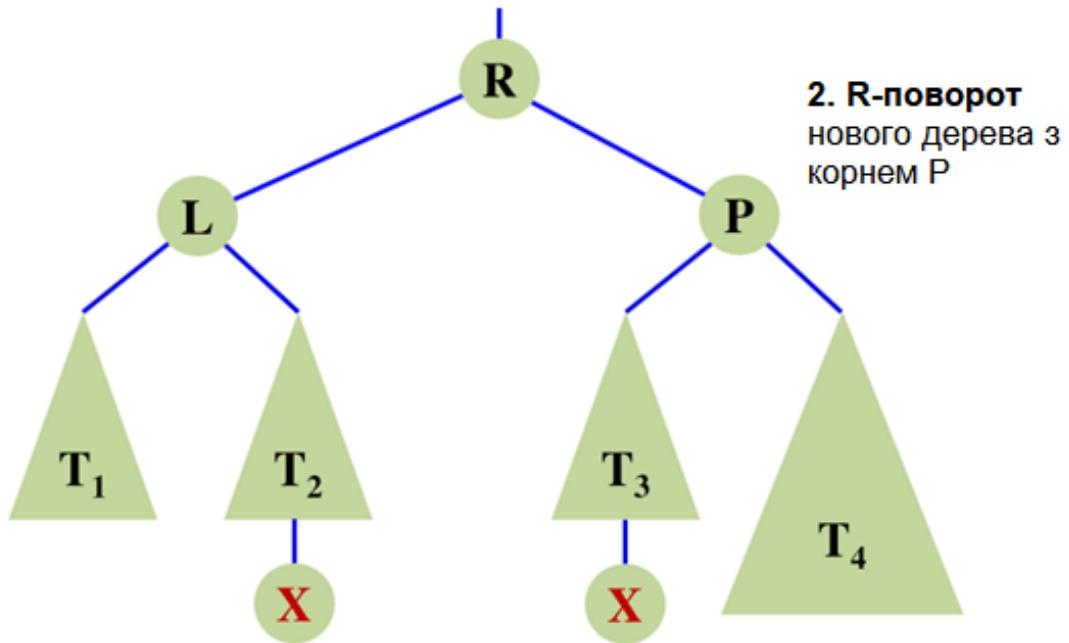
LR-поворот виконується після додавання елемента в праве піддерево лівого дочірнього вузла



LR-поворот виконується після додавання елемента в праве піддерево лівого дочірнього вузла

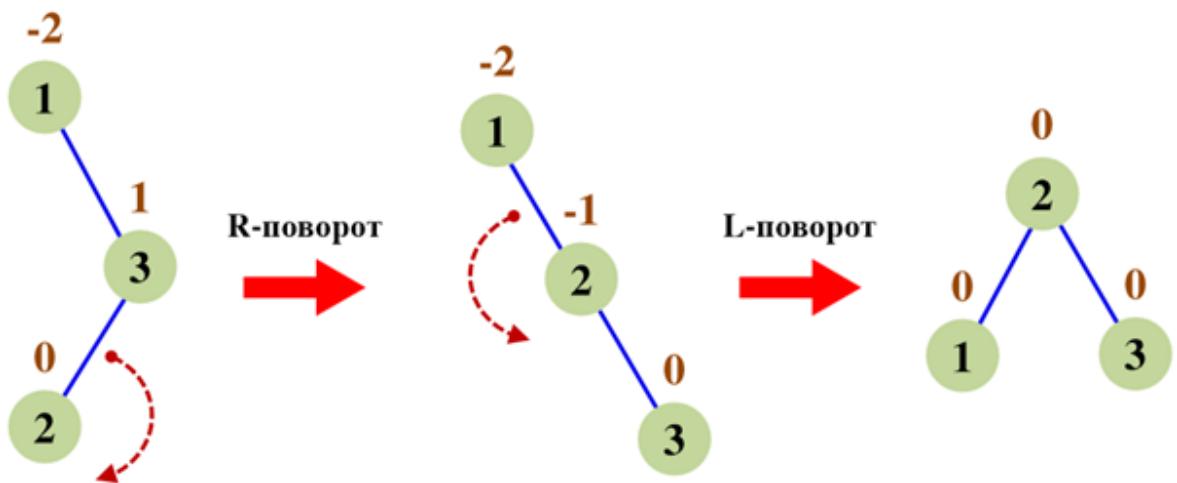


LR-поворот виконується після додавання елемента в праве піддерево лівого дочірнього вузла



Подвійний право-лівий поворот (RL-rotation, double right-left rotation)

RL-поворот виконується після додавання елемента в ліве піддерево правого дочірнього вузла



Будь-який поворот виконується за константний час - обчислювальна складність $O(1)$.

Будь-який поворот зберігає властивості бінарного дерева пошуку (розподіл ключів по лівим і правим піддеревам).

Алгоритм додавання вершини

Безпосередньо при додаванні листу присвоюється нульовий баланс. Процес включення вершини складається з трьох частин (даний процес описаний Ніклаус Віртом в «Алгоритми і структури даних»):

Крок 1. Прохода по шляху пошуку, поки не переконаємося, що ключа в дереві немає.

Крок 2. Включення нової вершини в дерево і визначення результуючих показників балансування.

Крок 3. «Відступ» назад по шляху пошуку і перевірки в кожній вершині показника збалансованості. Якщо необхідно - балансування.

Будемо повертати в якості результату функції, зменшилася висота дерева чи ні. Припустимо, що процес з лівої галузі повертається до батькові чи матері (рекурсія йде назад), тоді можливі три випадки: { h_l - висота лівого піддерева, h_r - висота правого піддерева} Включення вершини в ліве піддерево призведе до:

- $h_l < h_r$: вирівнюється $h_l = h_r$. Нічого робити не треба.
 - $h_l = h_r$: зараз ліве піддерево буде більше на одиницю, але балансировка поки що не потрібна.
 - $h_l > h_r$: зараз $h_l - h_r = 2$, – потрібна балансировка.
- У третьій ситуації потрібно визначити балансування лівого піддерева.

Алгоритм видалення вершини

Для простоти опишемо рекурсивний алгоритм видалення.

Крок 1. Якщо вершина - лист, то видалимо її і викличемо балансування всіх її предків в порядку від батька до кореня.

Крок 2. Інакше знайдемо найближчу за значенням вершину в поддереві найбільшої висоти (правому або лівому) і перемістимо її на місце видаляємої вершини, при цьому викликавши процедуру її видалення.

Доведемо, що даний алгоритм зберігає балансування. Для цього доведемо по індукції по висоті дерева, що після видалення деякої вершини з дерева і

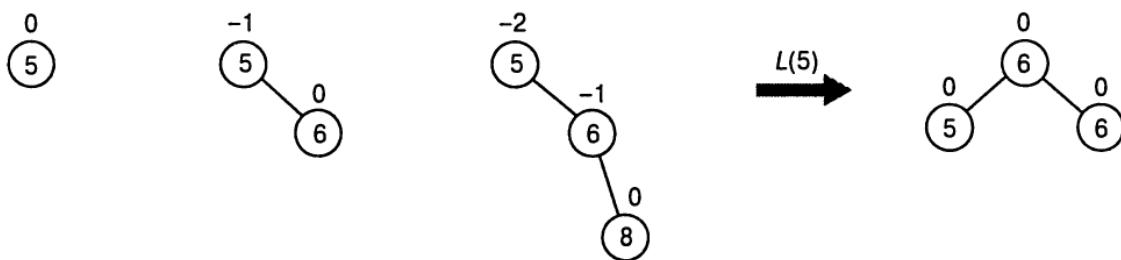
наступного балансування висота дерева зменшується не більше, ніж на 1. База індукції: Для листа очевидно вірно. Крок індукції: Або умова балансування в корені (після видалення корінь може змінитися) не порушуючи, тоді висота даного дерева не змінилася, або зменшилася строго менше з піддерев = \Rightarrow висота до балансування не змінилася = \Rightarrow після зменшиться не більше ніж на 1.

Очевидно, що в результаті зазначених дій процедура видалення викликається не більше 3 разів, так як у вершини, що видаляється з іншого виклику, немає одного з піддерев. Але пошук найближчого кожен раз вимагає $O(N)$ операцій. Стає очевидною можливість оптимізації: пошук найближчої вершини може бути виконаний по краю піддерева, що скорочує складність до $O(\log(N))$.

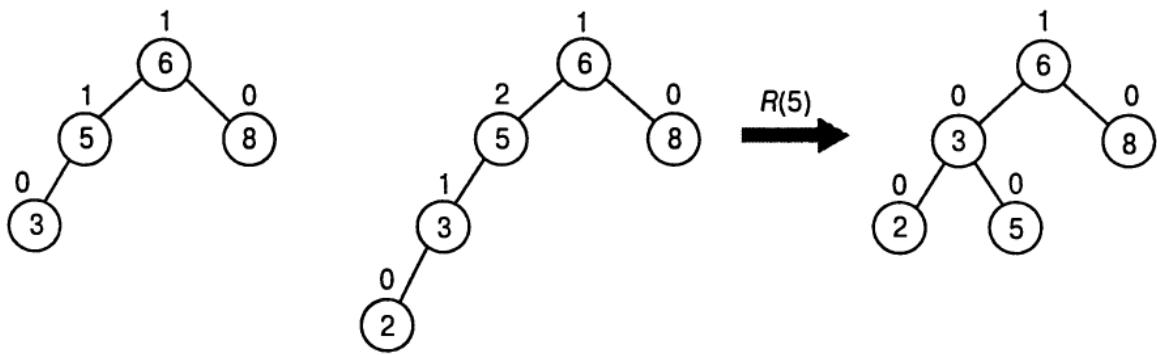
Приклад додавання елементів

Побудуємо AVL-дерево, шляхом послідовного додавання ключів: 5, 6, 8, 3, 2, 4, 7.

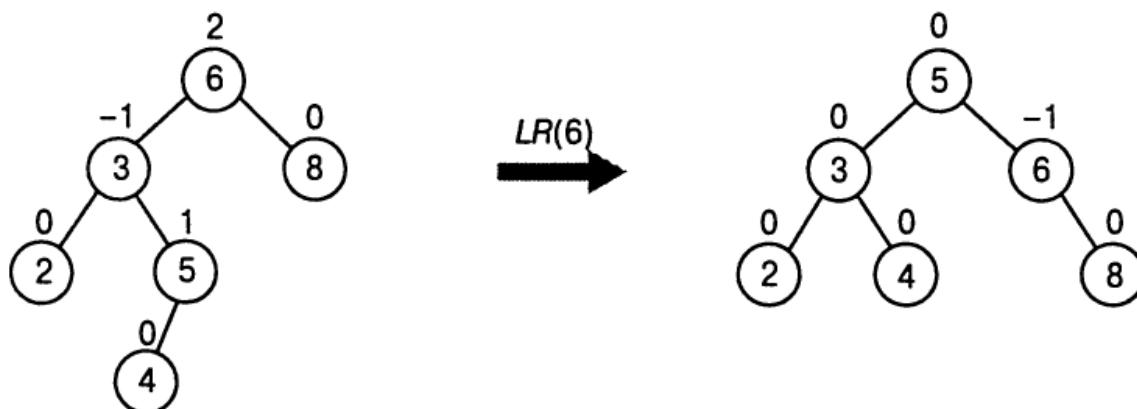
Комбінація балансів -2 -1 0 - означає **одиночний лівий поворот** (L-rotation, single left rotation)



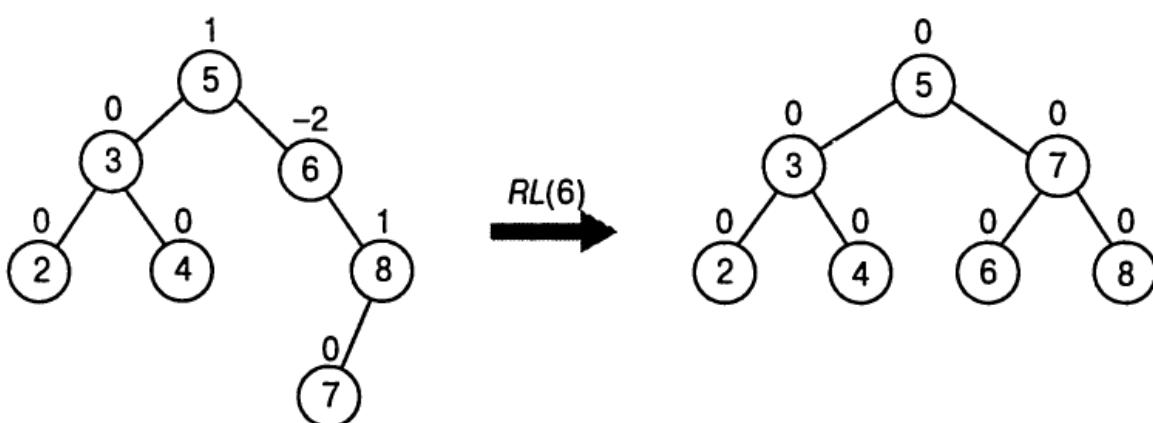
Комбінація балансів 2 1 0 - означає **одиночний правий поворот** (R-rotation, single right rotation).



Комбінація балансів 2 -1 0 - означає **подвійний ліво-правий поворот** (LR-rotation, double left-right rotation).



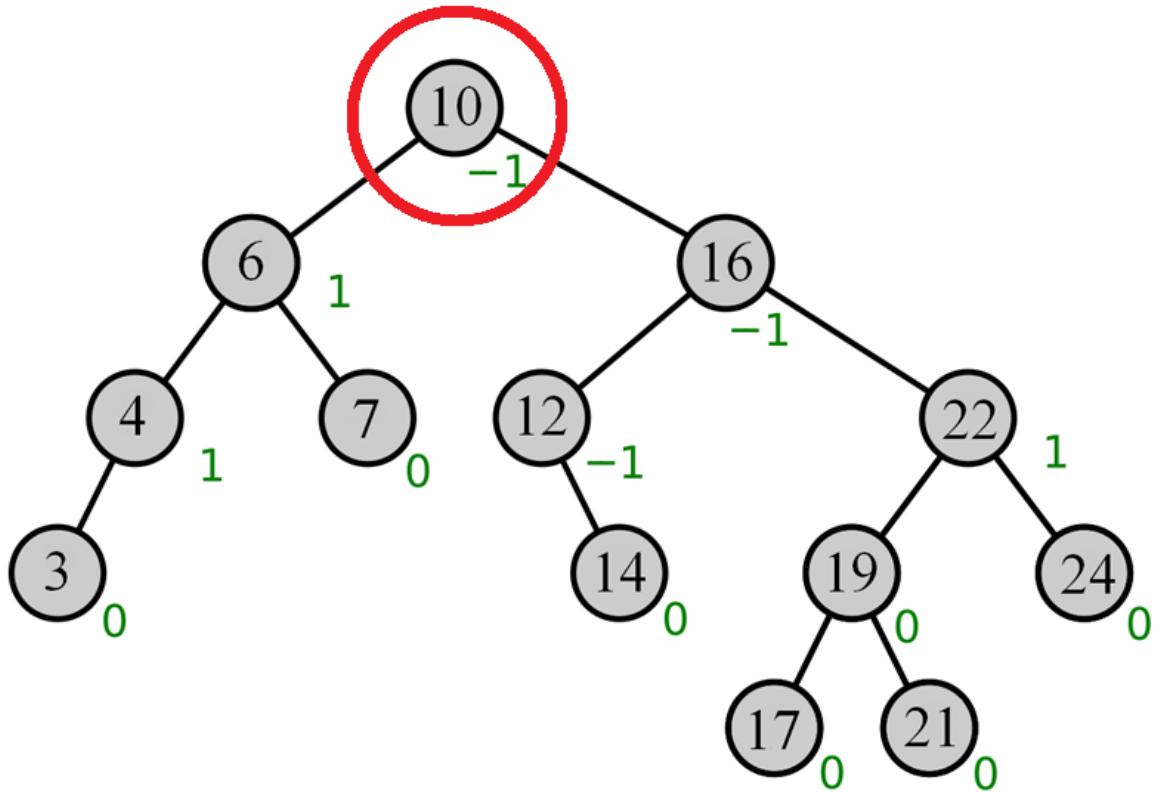
Комбінація балансів -2 1 0 – означає **подвійний ліво-правий поворот** (RL-rotation, double right-left rotation).



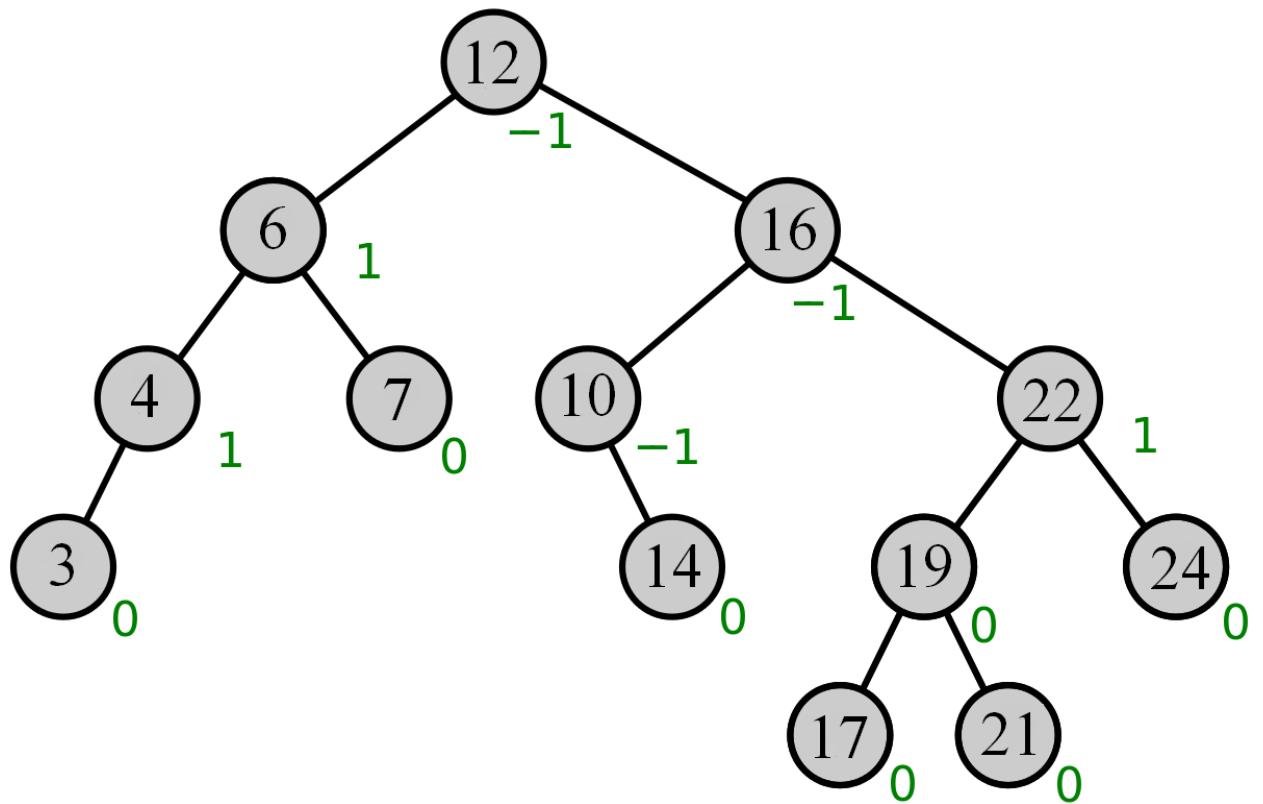
Числа в дужках після вказівки типу повороту - корінь дерева, що перебудовується.

Приклад видалення елементів

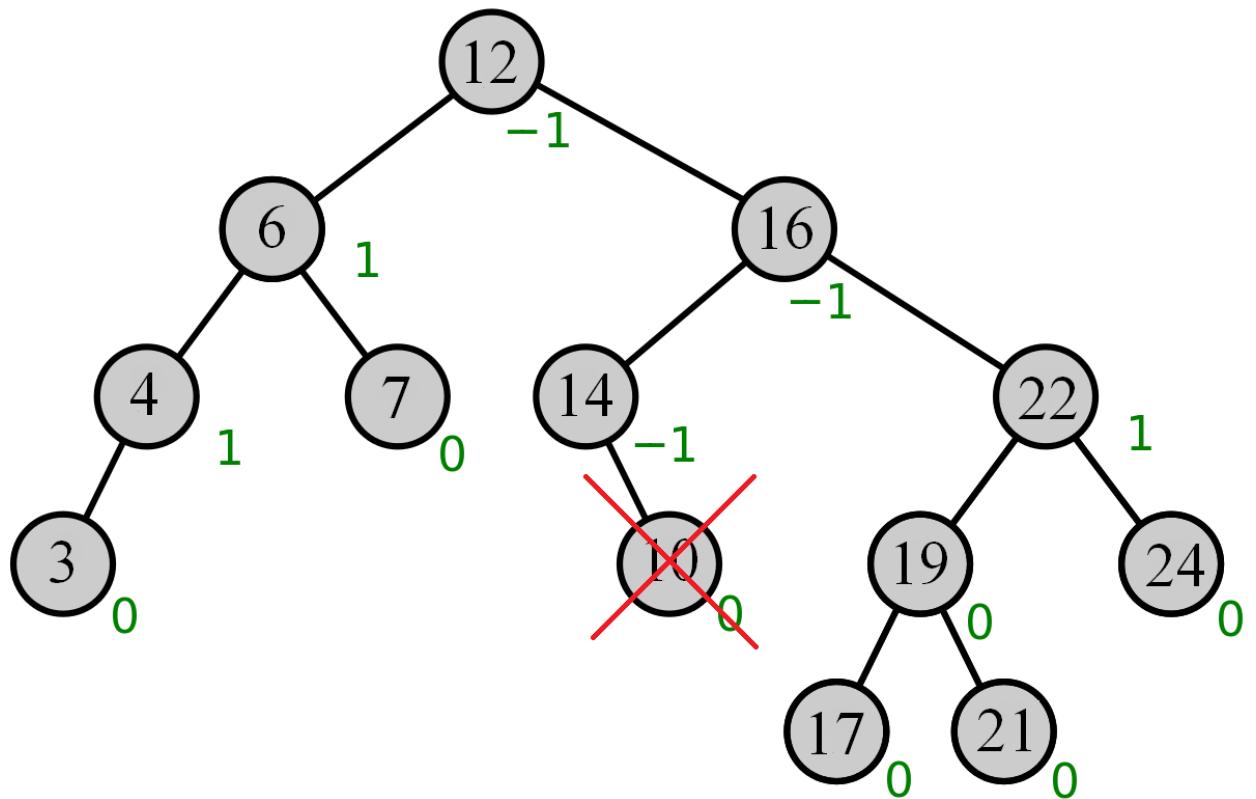
Видалимо ключ 10.



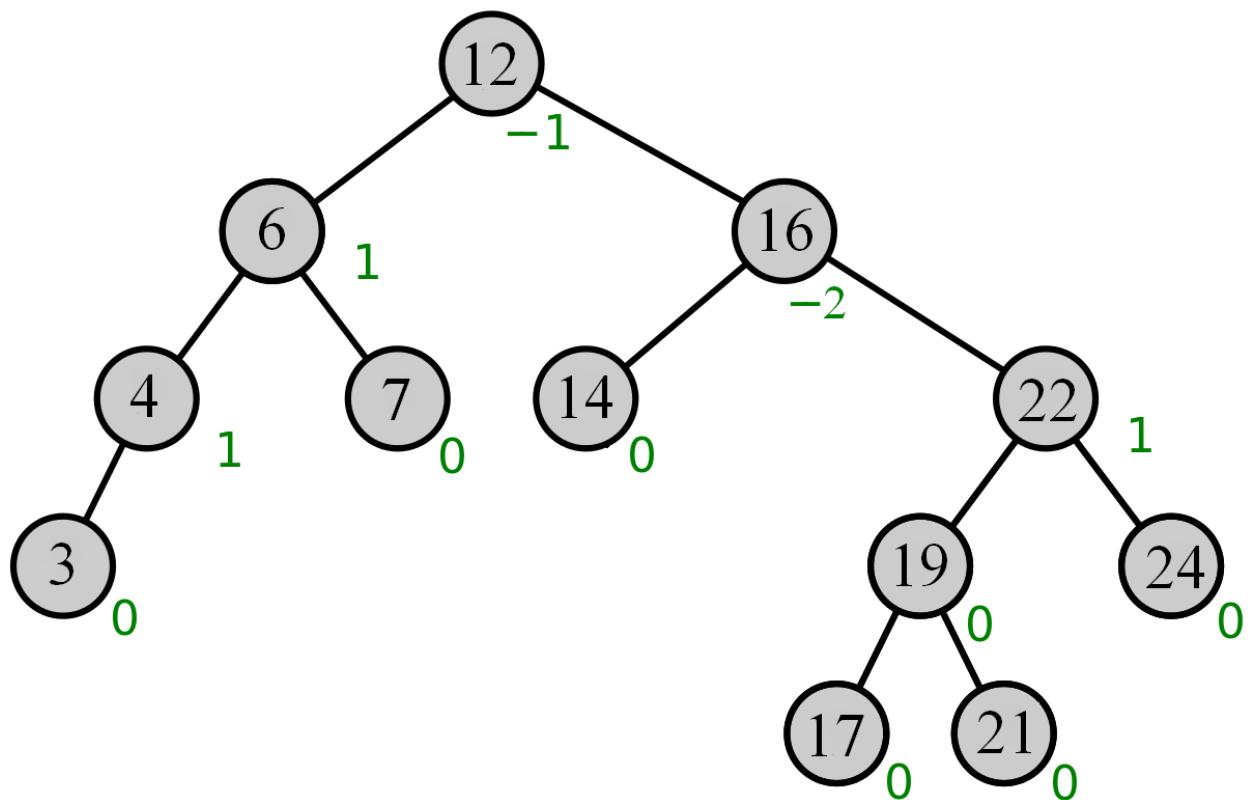
Розглянемо значення балансу в вершині з ключем 10, він дорівнює -1, тому розглядаємо праве піддерево (якщо 1 - ліве, якщо 0 - то в якому дочірній ключ більше за значенням), в ньому найближчий за значенням ключ до 10, дорівнює 12. Виконаємо заміну ключів 10 на 12.



Оскільки вершина з ключем 12 не була листом - необхідно повторити процедуру. Розглядаємо значення балансу в вершині з ключем 10, він дорівнює -1, тому розглядаємо праве піддерево, в ньому найближчий за значенням ключ до 10, дорівнює 14. Виконаємо заміну ключів 10 на 14. І тепер видалимо 10 як лист.

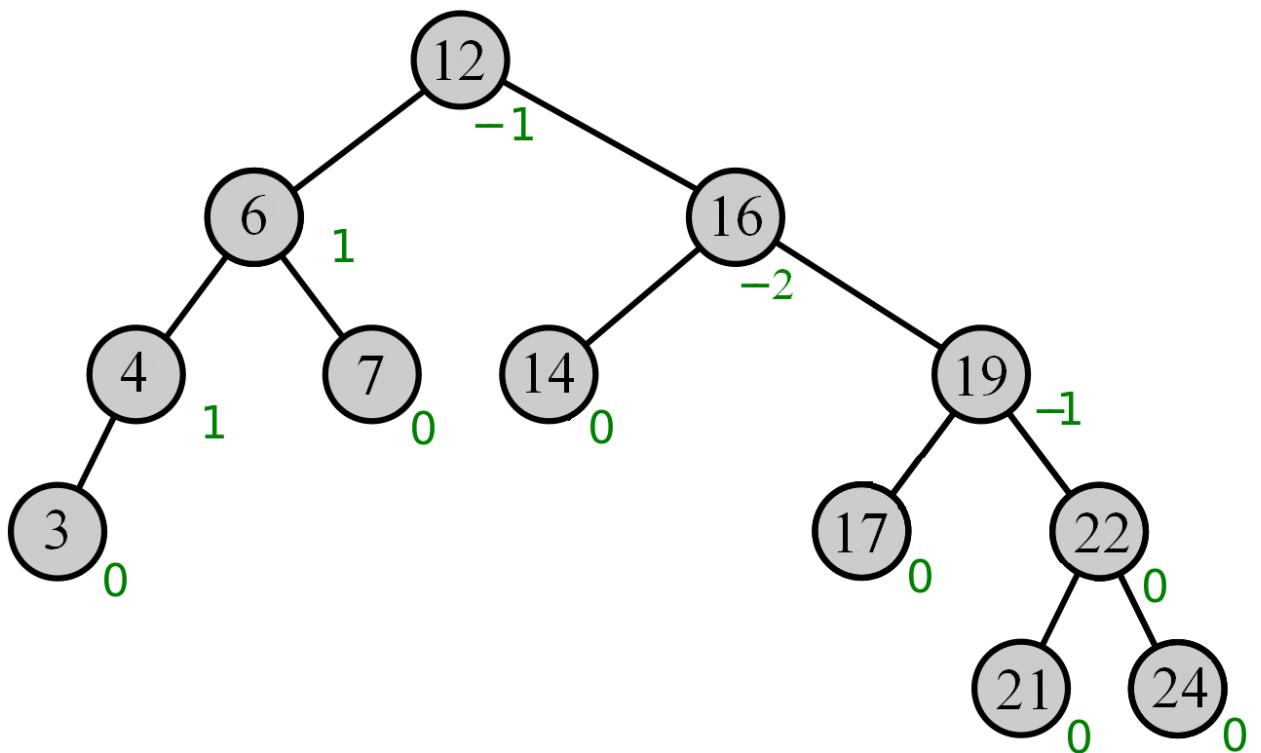


Перерахував баланси, бачимо, що необхідна балансировка.

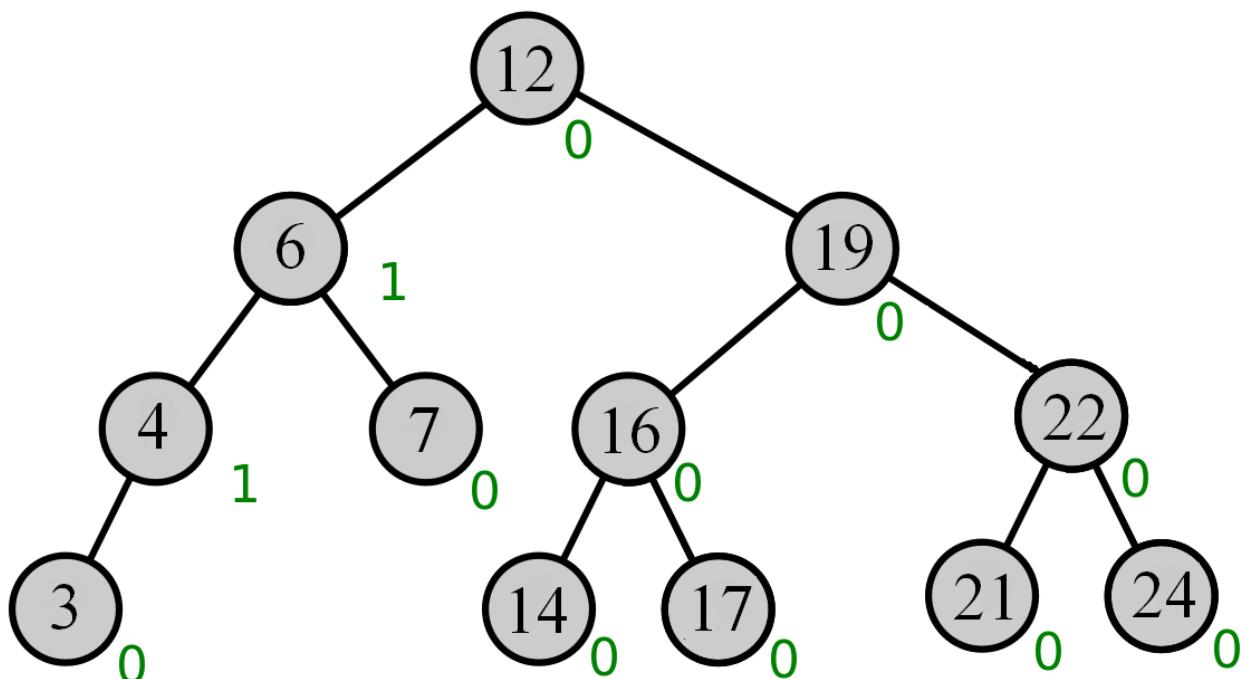


Комбінація балансів -2 1 0 - означає подвійний право-лівий поворот (RL-rotation, double right-left rotation).

Виконаємо правий поворот.



I лівий.



Складність

Операція	Средний случай (average case)	Худший случай (worst case)
Add (key, value)	$O(\log n)$	$O(\log n)$
Lookup (key)	$O(\log n)$	$O(\log n)$
Remove (key)	$O(\log n)$	$O(\log n)$
Min	$O(\log n)$	$O(\log n)$
Max	$O(\log n)$	$O(\log n)$

3.3 Червоно-чорні дерева

Червоно-чорне дерево (англ. Red-black tree, RB-Tree) являє собою бінарне дерево пошуку з одним додатковим бітом кольору в кожному вузлі. Колір вузла може бути або червоним, або чорним. Відповідно до обмежень, що накладаються на вузли дерева, шлях у червоно-чорному дереві не відрізняється від іншого по довжині більш раза у два рази, тому червоно-чорні дерева є приблизно збалансованими.

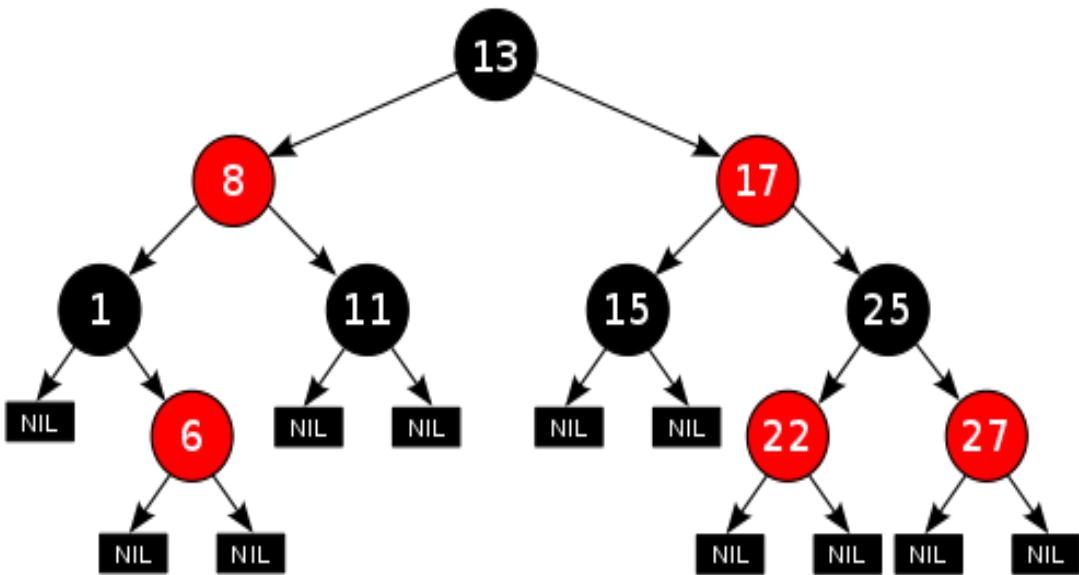
Кожен вузол дерева тепер містить атрибути *color*, *key*, *left*, *right* і *p*. Якщо дитина або батько вузла не існують, відповідний атрибут-вказівник цього вузла містить значення NIL. Ці NIL можна розглядати як вказівники на листи (зовнішні вузли) двійкового дерева пошуку, а звичайні вузли, що містять ключі, — як внутрішні вузли дерева.

Бінарне дерево пошуку є червоно-чорним деревом, якщо воно задовольняє наступним *червоно-чорним властивостям*:

1. Кожен вузол є червоним чи чорним.
2. Корінь дерева є чорним.
3. Кожен лист дерева (NIL) є чорним.
4. Якщо вузол — червоний, то обоє його дочірніх вузла — чорні.

5. Для кожного вузла всі шляхи від нього до листів, що є нащадками даного вузла, містять те саме кількість чорних вузлів.

Завдяки цим обмеженням, шлях від кореня до найдальшого листа не більше ніж удвічі довше, ніж до самого ближнього і дерево приблизно збалансовано.

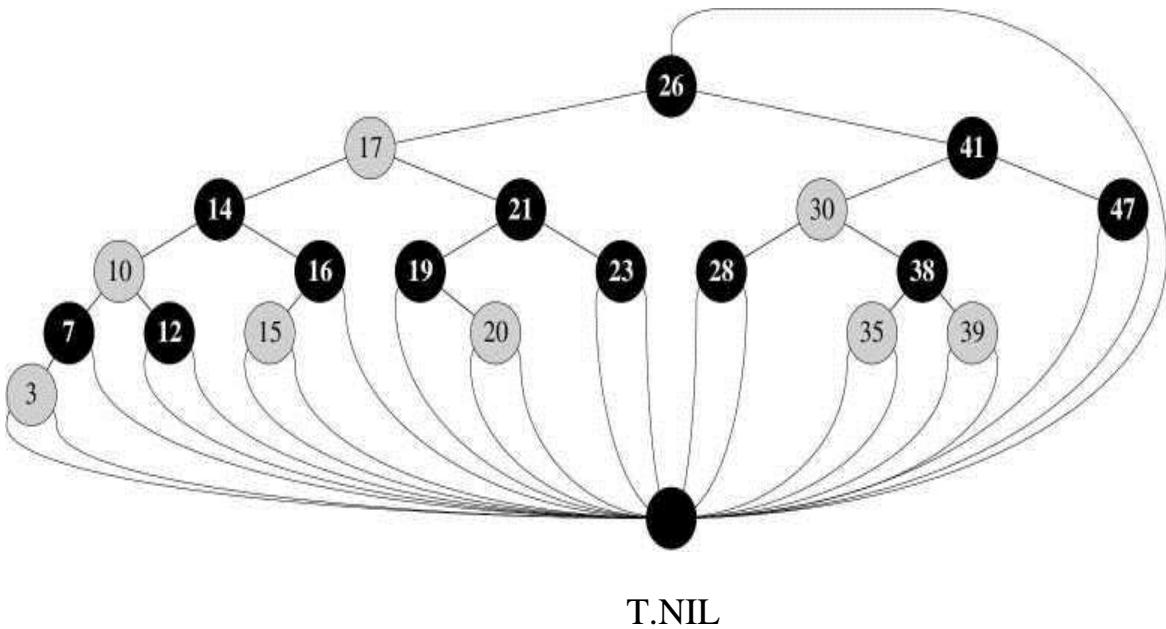


Число чорних вузлів уздовж будь-якого простого шляху від вузла x (не враховуючи сам вузол x) вниз до листа називають чорною висотою цього вузла і позначають $bh(x)$. Поняття чорної висоти однозначне, бо, за властивістю 5, усі низхідні прості шляхи від заданого вузла мають однакове число чорних вузлів. Чорну висоту червоно-чорного дерева означають як чорну висоту його кореня.

Для зручності роботи з крайовими умовами в червоно-чорному дереві предста-вимо NIL єдиним сторожовим об'єктом. У червоно-чорному дереві T сторож $T.nil$ — це об'єкт з тими ж атрибутами, що і звичайний вузол у дереві. Його атрибут `color` завжди містить *чорний*, а всі інші атрибути — `p`, `left`, `right` і `key` — можуть мати довільні значення. Як показано на рисунку 13.1(6), всі вказівники на NIL замінюються вказівником на сторожа $T.nil$.

Сторожовий об'єкт використано для того, щоб можна було розглядати дитину NIL вузла x як звичайний вузол, чий батько — x . Хоча можна додати різні сторожові вузли для кожного NIL у дереві, так що батько кожного NIL буде чітко визначений, але такий підхід марнуватиме пам'ять. Натомість використано одного сторожа $T.nil$ для представлення всіх NIL — усіх листів і

батька кореня. Значення атрибутів сторожа `p`, `left`, `right` і `key` неважливі, їх можна використовувати під час роботи процедури як буде зручно.

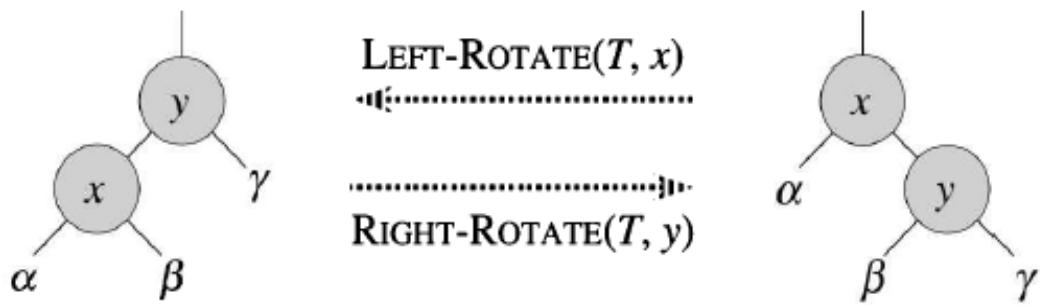


T.NIL

Повороти

Операції вставки, видалення і пошуку вимагають в гіршому випадку часу, пропорційного довжині дерева, що дозволяє червоно-чорним деревам бути більш ефективними в гіршому випадку, ніж звичайні двійкові дерева пошуку. Основні операції динамічних множин (вставка і видалення) виконуються за час $O(\lg n)$ у найгіршому випадку. Оскільки вони змінюють дерево, в результаті їх роботи можуть порушуватися червоно-чорні властивості. Для відновлення цих властивостей необхідно змінити кольори деяких вузлів дерева, а також структуру його покажчиків.

Структуру вказівників змінюють за допомогою *повороту* — локальної операції в дереві пошуку, яка зберігає властивість двійкового дерева пошуку. Використовується два види поворотів: лівий і правий.



У псевдокоді процедури LEFT-ROTATE припускається, що $x.right \neq T.nil$ і батько кореня — $T.nil$.

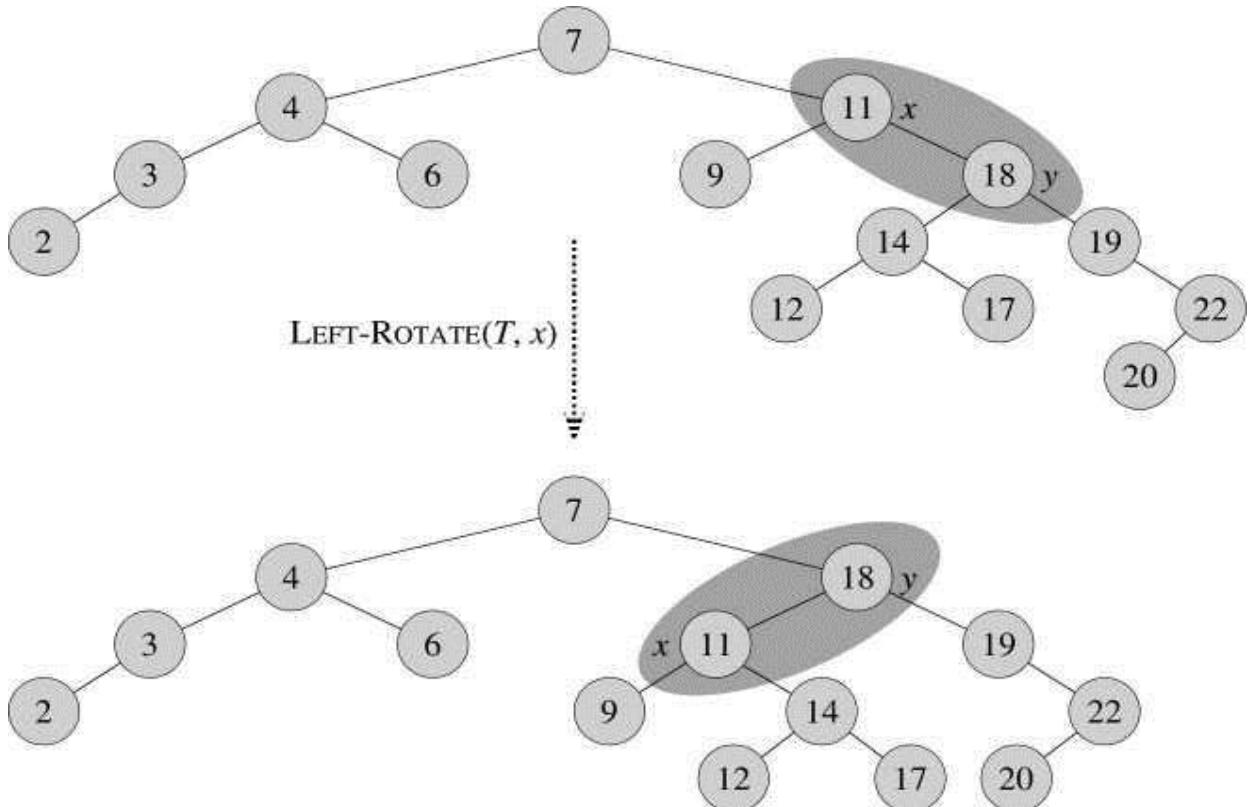
Left_Rotate(T, x)

```

1 y ← right[x] // Установлюємо у
2 right[x] ← left[y] // Ліве піддерево у стає правим піддеревом х
3 p[left[y]] ← x
4 p[y] ← p[x] // Перенос батька х у у
5 if p[x] = NULL[T]
6   then root[T] ← y
7   else if x = left[p[x]]
8     then left[p[x]] ← y
9     else right[p[x]] ← y
10 left[y] ← x // х - лівий дочірній у
11 p[x] ← y

```

На рисунку 13.3 показано приклад того, як процедура LEFT-ROTATE змінює двійкове дерево пошуку. Код RIGHT-ROTATE симетричний. Обидві процедури LEFT- ROTATE і RIGHT-ROTATE виконуються за час $O(1)$. Поворот змінює лише вказівники, всі інші атрибути вузла залишаються незмінними.



Вставка

Новий вузол в червоно-чорне дерево додається на місце одного з листя, забарвлюється в червоний колір і до нього прикріплюється два листа (так як листя є абстракцією, що не містить даних, їх додавання не вимагає додаткової операції). Що відбувається далі, залежить від кольору сусідніх вузлів. Зауважимо, що:

- Властивість 3 (Все листя чорні) виконується завжди.
- Властивість 4 (Обидва нащадка будь-якого червоного вузла - чорні) може порушитися тільки при додаванні червоного вузла, при перефарбування чорного вузла в червоний або при повороті.
- Властивість 5 (Всі шляхи від будь-якого вузла до листових вузлів містять однакове число чорних вузлів) може порушитися тільки при додаванні чорного вузла, перефарбування червоного вузла в чорний (або навпаки), або при повороті.

Примітка: Буквою N будемо позначати поточний вузол (пофарбований червоним). Спочатку це новий вузол, який вставляється, але ця процедура може застосовуватися рекурсивно до інших вузлів. Р будемо позначати «предка» N, через G позначимо «дідуся» N, а U будемо позначати «дядю» (вузол, що має загального батька з вузлом P). Відзначимо, що в деяких випадках ролі вузлів

можуть змінюватися, але, в будь-якому випадку, кожне позначення представлятиме той же вузол, що і на початку.

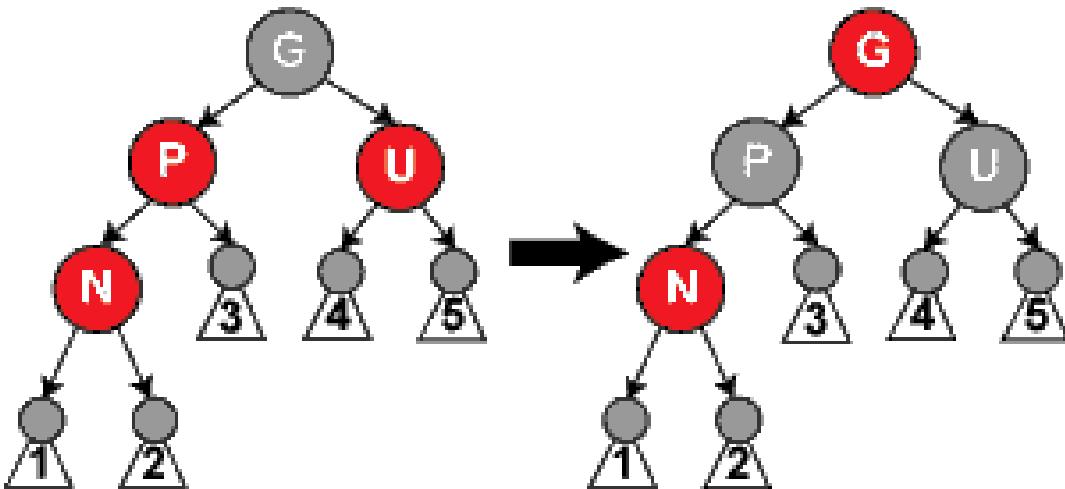
Далі можливий 1 з 5 випадків:

Випадок 1: Поточний вузол N в корені дерева. У цьому випадку, він перефарбовується в чорний колір, щоб виконувалася Властивість 2 (Корінь - чорний). Так як ця дія додає один чорний вузол в кожен шлях, Властивість 5 (Всі шляхи від будь-якого даного вузла до листових вузлів містять однакове число чорних вузлів) не порушується.

Випадок 2: Предок P поточного вузла чорний, тобто Властивість 4 (Обидва нащадка кожного червоного вузла - чорні) не порушується. У цьому випадку дерево залишається коректним. Властивість 5 (Всі шляхи від будь-якого даного вузла до листових вузлів містять однакове число чорних вузлів) не порушується, тому що поточний вузол N має двох чорних листових нащадків, але так як N є корисним, шлях до кожного з цих нащадків містить таке ж число чорних вузлів, що і шлях до чорного листа, який був замінений поточним вузлом, так що властивість залишається вірною.

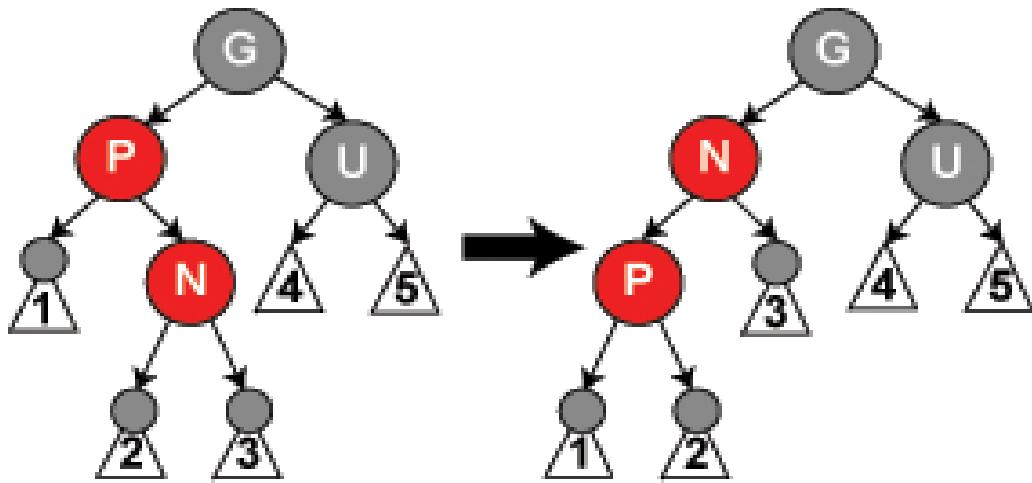
Примітка: У таких випадках передбачається, що у N є дідусь G , так як його батько P є корисним, а якби він був коренем, то був би пофарбований в чорний колір. Таким чином, N також має дядька U , хоча він може бути листовим вузлом у випадках 4 і 5.

Випадок 3: Якщо і батько P , і дядько U - червоні, то вони обидва можуть бути перефарбовані в чорний, і дідусь G стане червоним (для збереження властивості 5 (Всі шляхи від будь-якого даного вузла до листових вузлів містять однакове число чорних вузлів)). Тепер у поточного червоного вузла N чорний батько. Так як будь-який шлях через батька або дядька повинен проходити через дідуся, число чорних вузлів в цих шляхах не зміниться. Однак, дідусь G тепер може порушити властивості 2 (Корінь - чорний) або 4 (Обидва нащадка кожного червоного вузла - чорні) (властивість 4 може бути порушене, так як батько G може бути червоним). Щоб це відправити, вся процедура рекурсивно виконується на G з випадку 1.

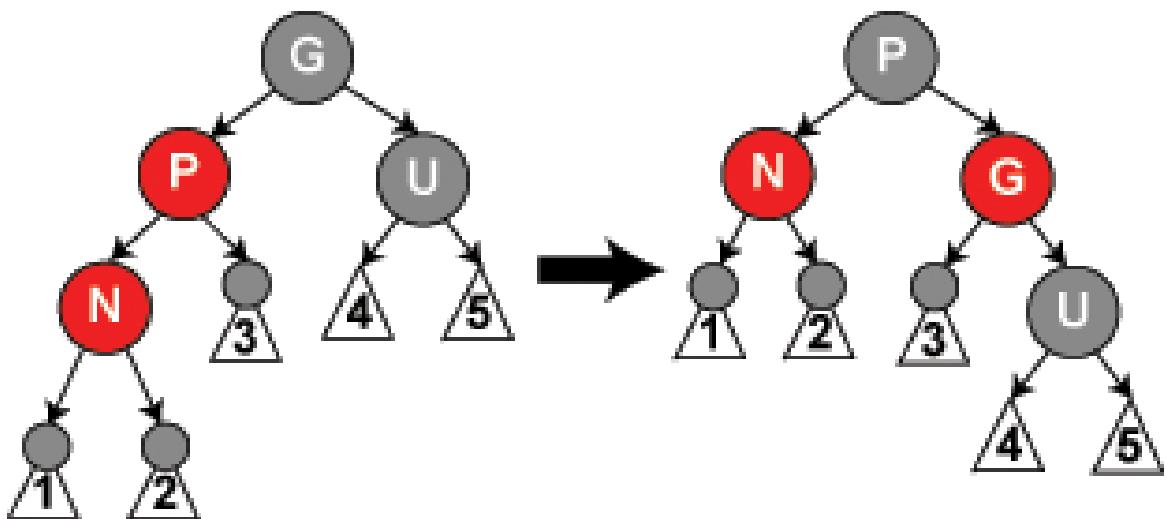


Примітка: У решти випадках передбачається, що батько P є лівим нащадком свого предка. Якщо це не так, треба щось міняти ліво і право (виконавши правий поворот).

Випадок 4: Батько P є червоним, але дядько U - чорний. Також, поточний вузол N - правий нащадок P, а P в свою чергу - лівий нащадок свого предка G. У цьому випадку може бути проведений лівий поворот дерева, який змінює ролі поточного вузла N і його предка P. Тоді, для колишнього батьківського вузла P в оновленій структурі використовуємо випадок 5, тому що Властивість 4 (Обидва нащадка будь-якого червоного вузла - чорні) все ще порушене. Обертання призводить до того, що деякі шляхи (в піддереве, позначеному «1» на схемі) проходять через вузол N, чого не було до цього. Це також призводить до того, що деякі шляхи (в піддереве, позначеному «3») не проходять через вузол P. Однак, обидва ці вузла є червоними, так що Властивість 5 (Всі шляхи від будь-якого даного вузла до листових вузлів містять однакове число чорних вузлів) не порушується при обертанні. Однак Властивість 4 все ще порушується, але тепер завдання зводиться до Випадку 5.



Випадок 5: Батько Р є червоним, але дядько U - чорний, поточний вузол N - лівий нащадок Р і Р - лівий нащадок G. У цьому випадку виконується правий поворот вузла G. У результаті виходить дерево, в якому колишній батько Р тепер є батьком і поточного вузла N і колишнього дідуся G. Відомо, що G - чорний, так як його колишній нащадок Р не міг би в іншому випадку бути червоним без порушення Властивості 4. Тоді кольори Р і G змінюються і в результаті дерево задоволяє Властивості 4 (Обидва нащадка будь-якого червоного вузла - чорні). Властивість 5 (Всі шляхи від будь-якого даного вузла до листових вузлів містять однакове число чорних вузлів) також залишається вірною, так як всі шляхи, які проходять через будь-який з цих трьох вузлів, раніше проходили через G, тому тепер вони всі проходять через Р. В кожному разі, з цих трьох вузлів тільки один забарвлений в чорний.



Дзеркальна ситуація. Батько Р є червоним, але дядько U - чорний, поточний вузол N - правий нащадок Р і Р - правий нащадок G. У цьому випадку виконується лівий поворот вузла G. Далі аналогічно.

Видалення

Варто зауважити, що новий видаляємий вузол не може мати відразу два дочірніх нелістових вузла, так як в протилежному випадку він не буде найбільшим / найменшим елементом. Таким чином, виходить, що випадок видалення вузла, що має два нелістових нащадка, зводиться до випадку видалення вузла, що містить як максимум один дочірній нелістовий вузол. Тому подальше опис буде виходити з припущення існування у видаляємого вузла не більше одного нелістового нащадка.

Будемо використовувати позначення M для видаляємого вузла; через C позначимо нащадка M, який також будемо називати просто «його нащадок». Якщо M має нелістового нащадка, візьмемо його за С. В іншому випадку за С візьмемо будь-який з листових нащадків.

Складним є випадок, коли і M і C - чорні. (Це може статися тільки тоді, коли видаляється чорний вузол, який має два листових нащадка, бо якщо чорний вузол M має чорного нелістового нащадка з одного боку, а з іншого - листового, то число чорних вузлів на обох сторонах буде різним і дерево стане недійсним червоно-чорним деревом через порушення Властивості 5.) Ми почнемо з заміни вузла M своїм нащадком С. Будемо називати цього нащадка (в своєму новому положенні) N, а його «брата» (іншого нащадка його нового предка) - S. (До цього S був «братом» M.) На малюнках нижче ми також будемо використовувати позначення P для нового предка N (старого предка M), SL для лівого нащадка S і SR для правого нащадка S (S не може бути листовим вузлом, так як якщо N за нашим припущенням є чорним, то дерево P, яке містить N, чорної висоти два і тому інше дерево P, яке містить S має бути також чорної висоти два, що не може бути в разі, коли S - лист).

Примітка: Для того, щоб дерево залишалося вірно визначенім, нам потрібно, щоб кожен лист залишався листом після всіх перетворень (щоб у

нього не було нащадків). Якщо видаляємий нами вузол має нелістового нащадка N , легко бачити, що властивість виконується. З іншого боку, якщо N - лист, то, як побачимо далі, властивість також виконується. Якщо N і його поточний батько чорні, тоді видалення батька призведе до того, що шляхи, які проходять через N матимуть на один чорний вузол менше, ніж шляху, які не проходять через нього. Так як це порушує властивість 5 (всі шляхи з будь-якого вузла до його листовим вузлів містять однакову кількість чорних вузлів), дерево повинно бути перебалансувати.

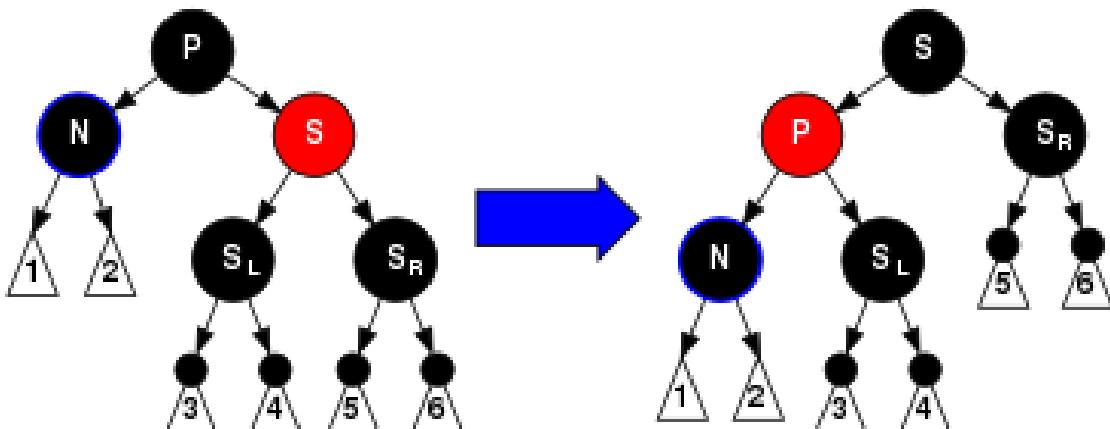
Є кілька випадків для розгляду:

Випадок 1: N - новий корінь. В цьому випадку, все зроблено. Ми видалили один чорний вузол з кожного шляху і новий корінь є чорним вузлом, тому властивості збережені.

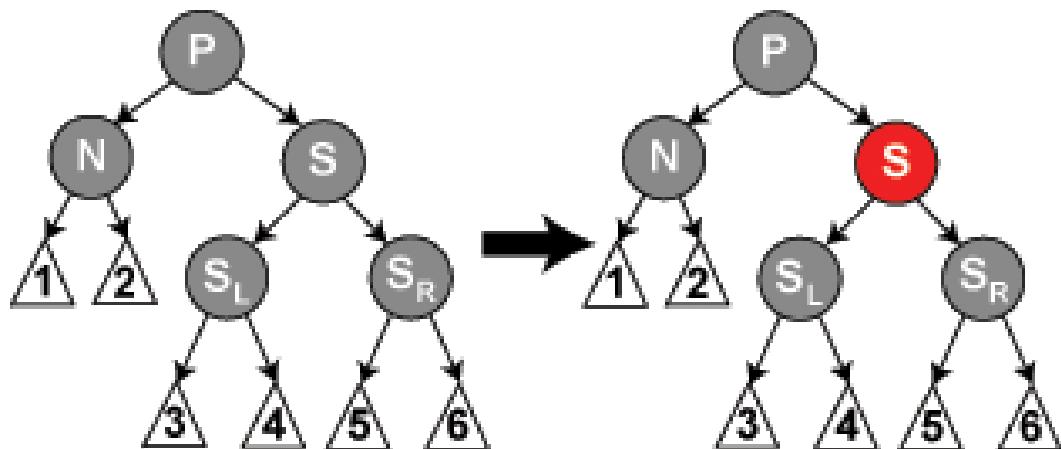
Примітка: У випадках 2, 5, і 6 ми припускаємо, що N є лівим нащадком свого предка P . Якщо він - правий нащадок, $left$ і $right$ потрібно поміняти місцями у всіх трьох випадках.

Випадок 2: S - червоний. В цьому випадку ми міняємо колір P і S , і потім робимо обертання вліво навколо P , ставлячи S дідусем N . Потрібно зауважити, що P має бути чорним, якщо він має червоного нащадка. Результатуюче піддерево все одно має чорних вузлів на одиницю менше, тому на цьому ми ще не закінчили. Тепер N має чорного брата і червоного батька, тому ми можемо перейти до кроку 4, 5 або 6. (Його новий брат є чорним тому, що він був нащадком червоного S .)

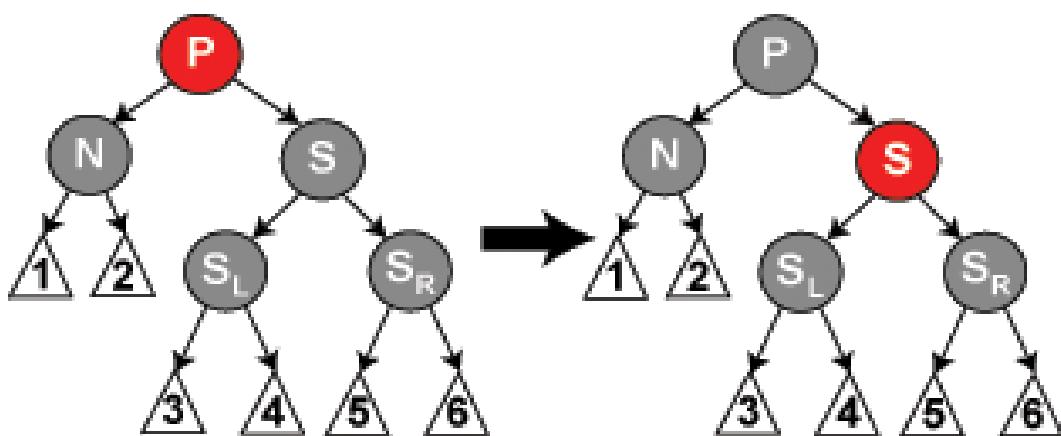
Примітка: Далі через S буде позначений новий брат N .



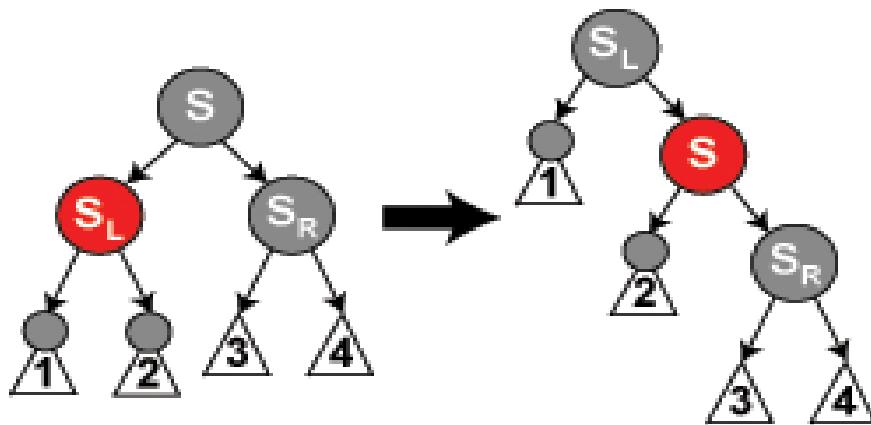
Випадок 3: P, S, і діти S - чорні. В цьому випадку ми просто перефарбовувати S в червоний. В результаті всі шляхи, що проходять через S, але не проходять через N, мають на один чорний вузол менше. Так як видалення батька N призводить до того, що всі шляхи, що проходять через N, містять на один чорний вузол менше, то такі дії вирівнюють баланс. Проте, всі, хто проходить через P шляху тепер містять на один чорний вузол менше, ніж шляху, які через P не проходять, тому властивість 5 (всі шляхи з будь-якої вершини до її листовим вузлів містять однакову кількість чорних вузлів) все ще порушене. Щоб це відправити, ми застосовуємо процедуру перебалансування до P, починаючи з випадку 1.



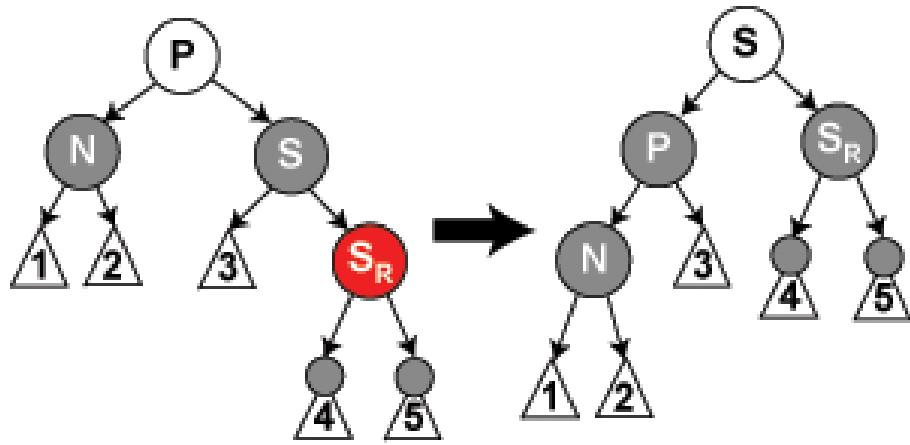
Випадок 4: S і його діти - чорні, але P - червоний. В цьому випадку ми просто змінюємо колір S і P. Це не впливає на кількість чорних вузлів на шляхах, що проходять через S, але додасть один до числа чорних вузлів на шляхах, що проходять через N, відновлюючи тим самим вплив віддаленого чорного вузла.



Випадок 5: S - чорний, лівий нащадок S - червоний, правий нащадок S - чорний, і N є лівим нащадком свого батька. В цьому випадку ми обертаємо дерево вправо навколо S. Таким чином лівий нащадок S стає його батьком і новим братом N. Після цього ми міняємо кольору у S і його нового батька. Всі шляхи, як і раніше містять однакову кількість чорних вузлів, але тепер у N є чорний брат з червоним правим нащадком, і ми переходимо до випадку 6. Ні N, ні його батько не впливають на цю трансформацію. (Для випадку 6 ми позначимо через S нового брата N.)



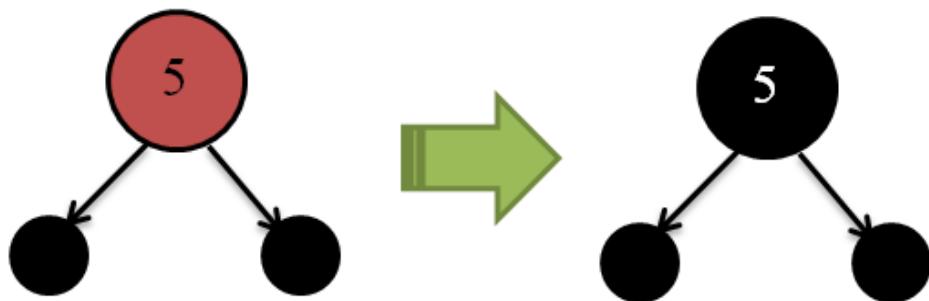
Випадок 6: S - чорний, правий нащадок S - червоний, і N є лівим нащадком свого батька P. У цьому випадку ми обертаємо дерево вліво навколо P, після чого S стає батьком P і свого правого нащадка. Далі ми міняємо місцями кольори у P і S (P приймає колір S, S приймає колір P), і робимо правого нащадка S чорним. Піддерево як і раніше має той же колір кореня, тому властивості 4 (Обидва нащадка кожного червоного вузла - чорні) і 5 (всі шляхи з будь-якої вершини до її листовим вузлів містять однакову кількість чорних вузлів) не порушуються. Проте, у N тепер з'явився додатковий чорний предок: або P став чорним, або він був чорним і S був доданий в якості чорного дідуся. Таким чином, що проходять через N шляху проходять через один додатковий чорний вузол.



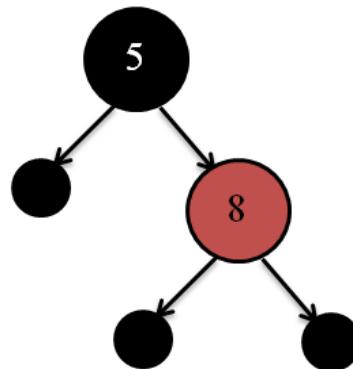
Приклад додавання елементів

Побудуємо червоно-чорне дерево, шляхом послідовного додавання ключів: 5, 8, 9, 10, 7, 6.

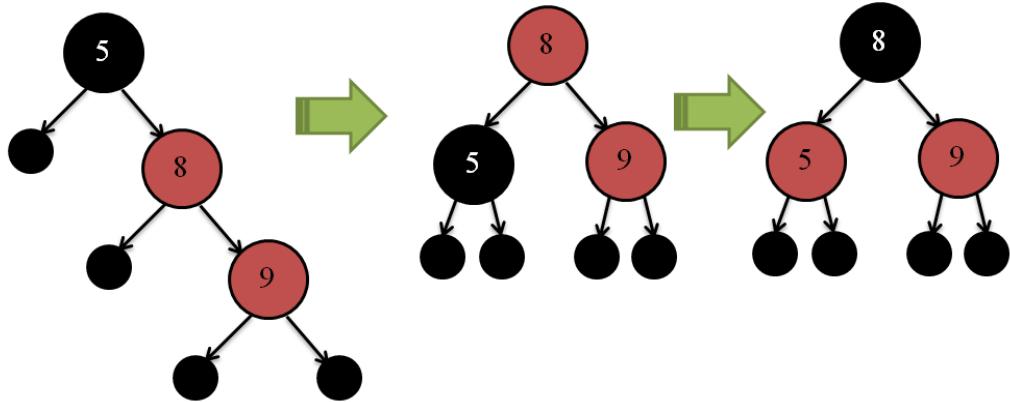
Так як елемент 5 є коренем - ми його просто перефарбовувати (Випадок 1).



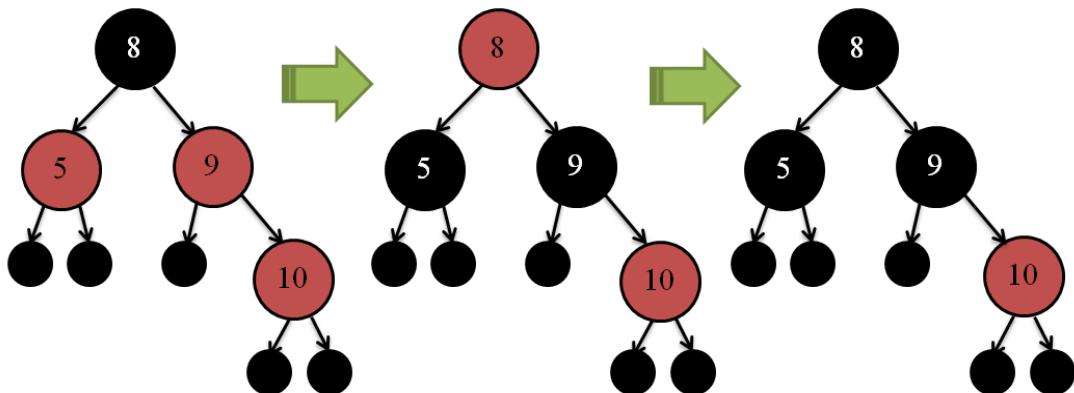
Додаємо елемент 8, властивості не порушуються (Випадок 2).



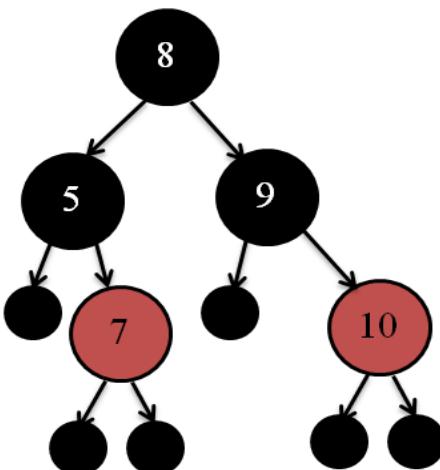
Додаємо елемент 9. Випадок 5, лівий поворот. Так як у нас дядько чорний, а дід і батько з одного боку, то ми застосовуємо 5 випадок - робимо поворот і перефарбовувати.



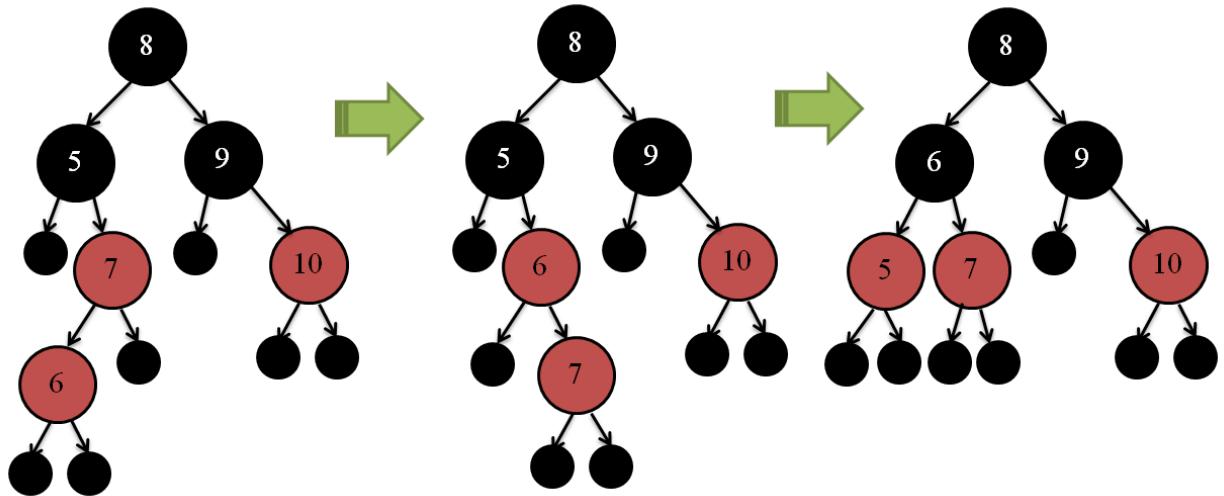
Додаємо елемент 10. Випадок 3 - перефарбували на початку 5 і 9 в чорний, а 8 в червоний, а потім 8 як корінь - в чорний. Якщо не корінь - просто міняємо кольору.



Додаємо елемент 7, властивості не порушуються (Випадок 2).



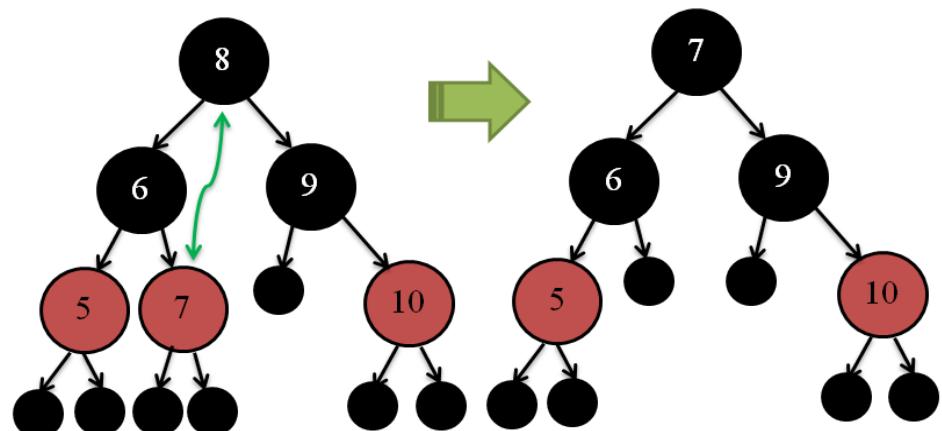
Додаємо елемент 6. Випадок 4, оскільки 6 - лівий нащадок, а 7 правий - необхідно виконати правий поворот, після чого ми отримаємо 5й випадок з лівим поворотом.



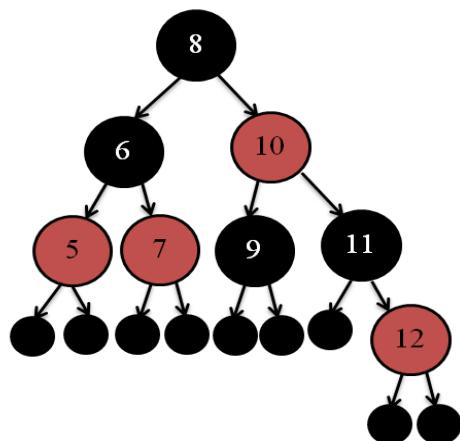
Приклад видалення елементів

Візьмемо кінцеве дерево з минулого прикладу і видалимо ключ 8.

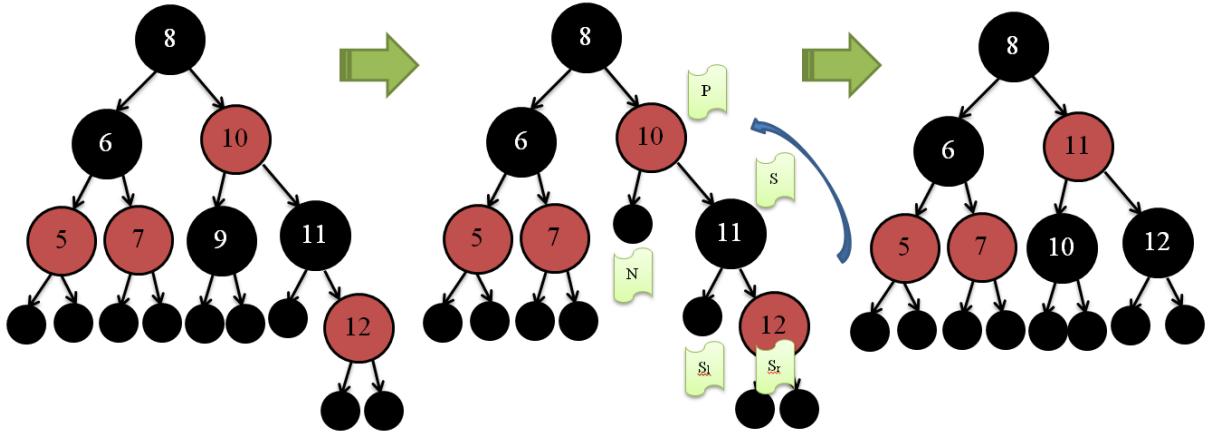
Випадок 1, після видалення ми отримали червоно-чорне дерево.



Нехай маємо наступне червоно-чоне дерево.



Видалимо ключ 9. Ключ 9 має 2 листових нащадка, видаливши його ми порушимо властивість 5, тому потрібно встановити властивості червоно-чорного дерева. Оскільки предок Р - червоний, брат S - чорний, а правий нащадок брата Sr - червоний, маємо випадок 6. Виконуємо лівий поворот ключа 10, 10 і 11 змінюють колір на протилежний, а Sr стає чорним.



Складність

Операція	Средний случай (average case)	Худший случай (worst case)
Add(key, value)	$O(\log n)$	$O(\log n)$
Lookup(key)	$O(\log n)$	$O(\log n)$
Remove(key)	$O(\log n)$	$O(\log n)$
Min	$O(\log n)$	$O(\log n)$
Max	$O(\log n)$	$O(\log n)$

3.4 В-дерево

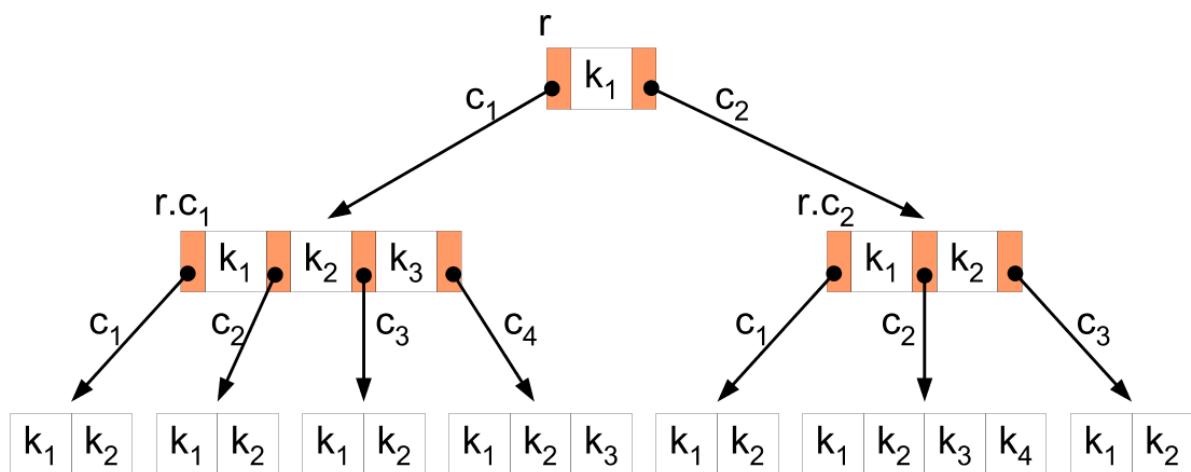
В-дерево (Бі-дерево) - структура даних, дерево пошуку. З точки зору зовнішнього логічного уявлення, збалансоване, сильно гіллясте дерево. Часто використовується для зберігання даних у зовнішній пам'яті.

Використання В-дерев вперше було запропоновано Р. Бейером і Е. МакКрейтом в 1970 році.

Збалансованість означає, що довжина будь-яких двох шляхів від кореня до листя розрізняється не більше, ніж на одиницю.

Гіллястість дерева - це властивість кожного вузла дерева посилатися на велика кількість вузлів-нащадків.

З точки зору фізичної організації В-дерево представляється як мультиспіскова структура сторінок пам'яті, тобто кожному вузлу дерева відповідає блок пам'яті (сторінка). Внутрішні і листові сторінки зазвичай мають різну структуру.



Застосування

Структура В-дерева застосовується для організації індексів у багатьох сучасних СУБД.

В-дерево може застосовуватися для структурування (індексування) інформації на жорсткому диску (як правило, метаданих). Час доступу до довільного блоку на жорсткому диску дуже велике (близько мілісекунд), оскільки воно визначається швидкістю обертання диска і переміщення головок. Тому важливо зменшити кількість вузлів, що переглядаються при кожній операції. Використання пошуку за списком кожен раз для знаходження випадкового блоку могло б привести до надмірної кількості звернень до диска внаслідок необхідності здійснення послідовного проходу по всіх його елементах, що передує заданому, тоді як пошук в В-дереві, завдяки

властивостям збалансованості і високою гіллястості, дозволяє значно скоротити кількість таких операцій.

Відносно проста реалізація алгоритмів і існування готових бібліотек (в тому числі для С) для роботи зі структурою В-дерева забезпечують популярність застосування такої організації пам'яті в найрізноманітніших програмах, що працюють з великими обсягами даних.

Структура і принципи побудови

В-деревом називається дерево, яке задовольняє наступним властивостям:

- Ключі в кожному вузлі зазвичай впорядковані для швидкого доступу до них. Корінь містить від 1 до $2t-1$ ключів. Будь-який інший вузол містить від $t-1$ до $2t-1$ ключів. Листя не є винятком з цього правила. Тут t - параметр дерева, не менший 2 (і зазвичай приймає значення від 50 до 2000).
- У листя нащадків немає. Будь-який інший вузол, що містить ключі K_1, \dots, K_n , містить $n + 1$ нащадків.
- Перший нащадок і все його нащадки містять ключі з інтервалу $(-\infty, K_1)$.
- Для $2 \leq i \leq n$, i -й нащадок і все його нащадки містять ключі з інтервалу (K_{i-1}, K_i) .
- $(N + 1)$ нащадок і всі його нащадки містять ключі з інтервалу (K_n, ∞)
- Глибина всіх листів однакова.

Кожний вузел В-дерева, окрім листя, можна розглядати як упорядкований список, у якому чергуються ключі і вказівники на нащадків.

Пошук

Якщо ключ міститься в корені, він знайдений. Інакше визначаємо інтервал і йдемо до відповідного нащадку. Повторюємо, поки не дійшли до листа.

Додавання ключа

Будемо називати деревом нащадків якогось вузла піддерево, що складається з цього вузла та його нащадків.

Спочатку визначимо функцію, яка додає ключ К до дерева нащадків вузла х. Після виконання функції у всіх пройдених вузлах, крім, може бути, самого вузла х, буде менше $2t - 1$, але не менше $t - 1$, ключів.

Якщо х – не лист,

- Визначаємо інтервал, де повинен знаходитися К. Нехай у – відповідний нащадок.
- Рекурсивно додаємо К до дерева нащадків у.
- Якщо вузол у сповнений, тобто містить $2t - 1$ ключів, розщеплюємо його на два. Вузол u_1 отримує перші $t - 1$ з ключів у і перші t його нащадків, а вузол u_2 – останні $t - 1$ з ключів у і останні t його нащадків. Медіанний з ключів вузла у потрапляє в вузол х, а покажчик на у у вузлі х замінюється показчиками на вузли u_1 і u_2 .

Якщо х – лист, лише додаємо туди ключ К.

Тепер визначимо додавання ключа К до всього дерева. Буквою R позначається кореневий вузол.

Додамо К до дерева нащадків R.

Якщо R містить тепер $2t-1$ ключів, розщеплюємо його на два. Вузол R_1 отримує перші $t-1$ з ключів R і перші t його нащадків, а вузол R_2 – останні $t-1$ з ключів R і останні t його нащадків. Медіанний з ключів вузла R потрапляє во новостворений вузол, який стає кореневим. Вузли R_1 і R_2 стають його нащадками.

Приклад

8, 12, 5, 0, 15, 7, 23, 48, 16, 51

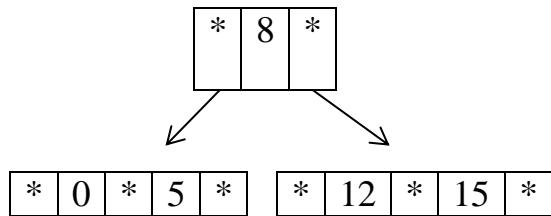
$t=3$

Поки кореневої вузол не буде містити $2t-1$ ключів, додаємо в нього ключі в упорядкованому вигляді.

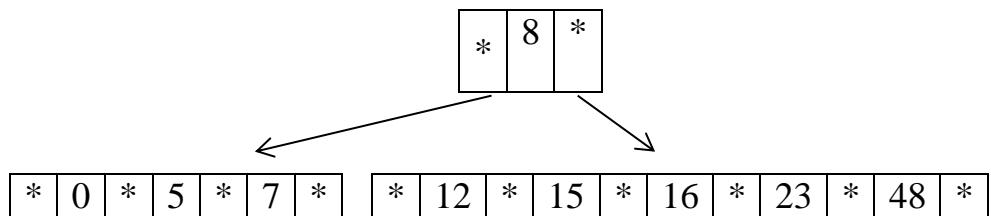
$$[\ast] \rightarrow [\ast | 8 | \ast] \rightarrow [\ast | 8 | \ast | 12 | \ast] \rightarrow [\ast | 5 | \ast | 8 | \ast | 12 | \ast]$$

$$[\ast | 0 | \ast | 5 | \ast | 8 | \ast | 12 | \ast] \quad [\ast | 0 | \ast | 5 | \ast | 8 | \ast | 12 | \ast | 15 | \ast]$$

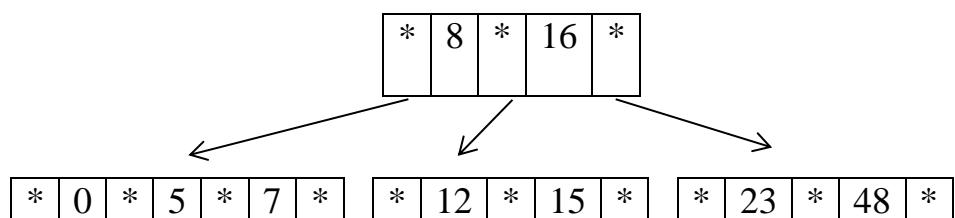
Коли кореневої вузол заповнений, для додавання наступного значення, нам необхідно розділити вузол на два по медианному ключу.



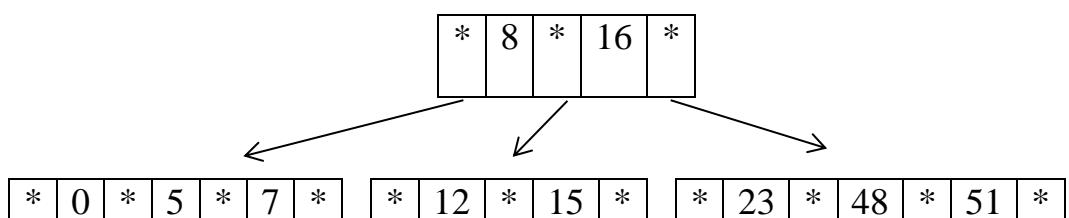
Додаємо ключі 7, 23, 48, 16.



Необхідно розщепити вузел.



Додаємо ключ 51.



Видалення ключа

Якщо корінь одночасно є листом, тобто в дереві всього один вузол, ми просто видаляємо ключ з цього вузла. В іншому випадку спочатку знаходимо вузол, що містить ключ, запам'ятовуючи шлях до нього. Нехай цей вузол - x.

Якщо ключ до знаходить у вузлі x і x є листом, видаляємо ключ k з x.

Якщо ключ k знаходиться у вузлі x і x є внутрішнім вузлом, робимо наступне.

а) Якщо дочірній по відношенню до x вузол y , що передує ключу k у вузлі x , містить не менше t ключів, то знаходимо k' -го попередника k в піддереві, коренем якого є y . Рекурсивно видаляємо k' і замінююємо k в x ключем k' . (Пошук ключа k і його видалення можна виконати за один спадний прохід.)

б) Якщо y має менш t ключів, то симетрично звертаємося до дочірнього по відношенню до x вузлу z , який слідує за ключем k у вузлі x . Якщо z містить не менше t ключів, то знаходимо k' -ий наступний за k ключ в піддереві, коренем якого є z . Рекурсивно видаляємо k' і замінююємо k в x ключем k' . (Пошук ключа k і його видалення можна виконати за один спадний прохід.)

в) В іншому випадку, якщо i у, i z містять по $t - 1$ ключів, вносимо k і всі ключі z в y (при цьому з x видаляються i k , i покажчик на z , а вузол y після цього містить $2t - 1$ ключів), а потім звільняємо z і рекурсивно видаляємо k з y .

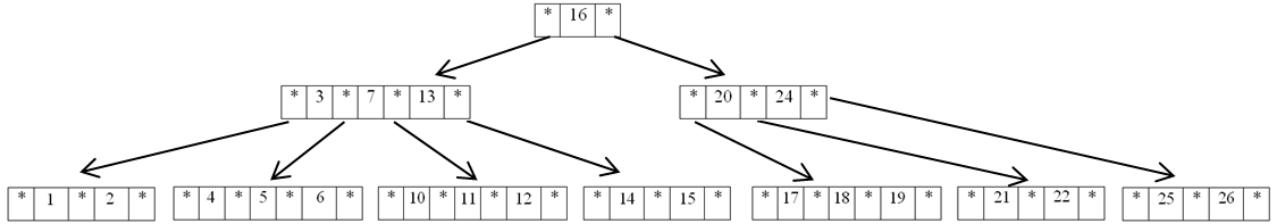
Якщо ключ k відсутній у внутрішньому вузлі x , знаходимо корінь $x.c_1$ піддерева, яке повинно містити k (якщо такий ключ є в даному B-дереві). Якщо $x.c_1$ містить тільки $t - 1$ ключів, виконуємо (а) або (б) для того, щоб гарантувати, що далі ми переходимо в вузол, що містить як мінімум t ключів. Потім ми рекурсивно видаляємо k з відповідного дочірнього по відношенню до x вузла.

а) Якщо $x.c_1$ має тільки $t - 1$ ключів, але при цьому один з його безпосередніх сусідів (під яким ми розуміємо дочірній по відношенню до x вузол, відокремлений від розглянутого рівно одним ключем-роздільником) містить як мінімум t ключів, передамо в $x.c_1$ ключ-роздільник між цим вузлом і його безпосереднім сусідом з x , на його місце помістимо крайній ключ з сусіднього вузла і перенесемо відповідний покажчик з сусіднього вузла в $x.c_1$.

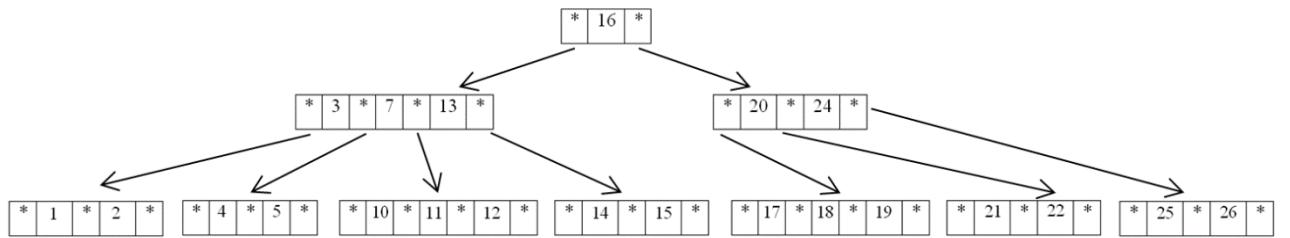
б) Якщо i $x.c_1$, i обидва його безпосередніх сусідів містять по $t - 1$ ключів, об'єднаємо $x.c_1$ з одним з його сусідів (при цьому колишній ключ-роздільник з x буде перенесений вниз і стане медіаною нового вузла).

Приклад:

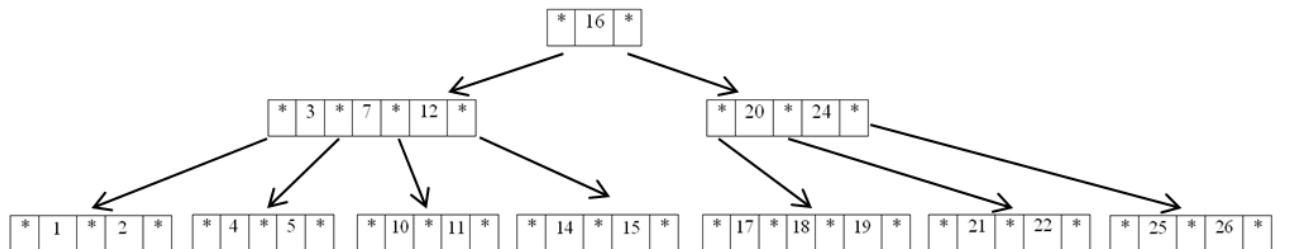
Видалимо ключі 6, 13, 7



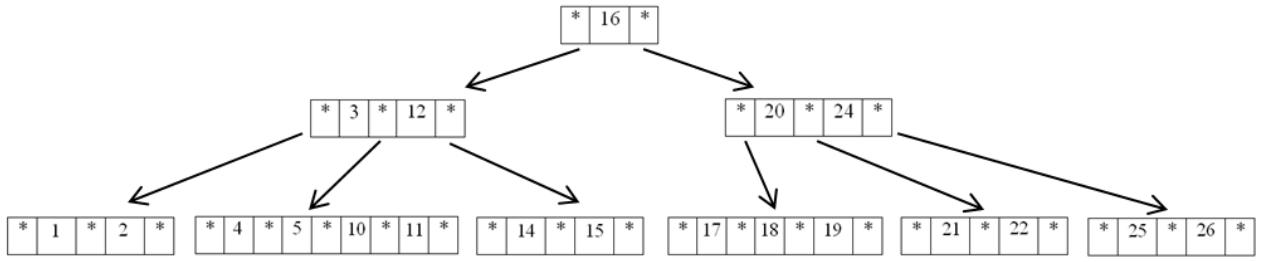
Видаляємо ключ 6. Ключ знаходиться в листі, видаляємо звідти ключ. У вузлі залишилося не менше $t-1$ ключів, ми на цьому зупиняємося.



Видаляємо ключ 13. Ключ знаходиться у внутрішньому вузлі, видаляємо найправіший ключ з піддерева нашадків для ключа 13, тобто ключ 12, замінюємо ключ 13 видаленим з листа.



Видаляємо ключ 7. Ключ знаходиться у внутрішньому вузлі, перепишемо ключі 7, 10 і 11 в вузол з ключами 4 і 5, після чого видалимо ключ 7 з внутрішнього вузла і покажчик на вузол 10, 11 разом з самим вузлом. Після цього видалимо ключ 7.



Основні переваги

У всіх випадках корисне використання простору вторинної пам'яті становить понад 50%. З ростом ступеня корисного використання пам'яті не відбувається зниження якості обслуговування.

Довільний доступ до запису реалізується за допомогою малої кількості підоперацій (звернення до фізичних блоків).

В середньому досить ефективно реалізуються операції включення та видалення записів; при цьому зберігається природний порядок ключів з метою послідовної обробки, а також відповідний баланс дерева для забезпечення швидкої довільної вибірки.

Незмінна впорядкованість по ключу забезпечує можливість пакетної обробки.

Основний недолік В-дерев полягає у відсутності для них ефективних засобів вибірки даних (тобто методу обходу дерева), упорядкованих по відмінному від обраного ключа.

4 ТЕМА 4 СУЧАСНІ ЕВРИСТИЧНІ АЛГОРИТМИ

Метаевристика - метод оптимізації, що багаторазово використовує прості правила або евристики для досягнення оптимального або субоптимального рішення.

Метаевристичні алгоритми - алгоритми, що реалізують прямий випадковий пошук можливих рішень задачі, оптимальних або близьких до оптимальних, поки не буде виконано якусь умову або досягнуто задане число ітерацій.

Метаевристичні алгоритми (метаевристики) є потужним і надзвичайно популярним класом оптимізаційних методів, що дозволяють знаходити рішення для широкого кола завдань з різних додатків. Їх сила полягає в їх здатності вирішення складних завдань без знання простору пошуку, саме тому ці методи дають можливість вирішувати важкорозв'язувані завдання оптимізації.

Клас метаевристичних алгоритмів включає в себе але не обмежується алгоритми оптимізації мурашиної колонії (ant colony optimization (ACO)), еволюційні обчислення, включаючи генетичні алгоритми (ГА), ітеративний локальний пошук, метод імітації відпалу і алгоритм табу-пошуку (або пошуку з заборонами).

Метаевристики це загальні евристики, що дозволяють знаходити близькі до оптимальних рішення різних завдань оптимізації за прийнятний час.

Різні описи метаевристик в літературі дозволяють сформулювати деякі фундаментальні властивості, якими характеризуються метаевристики:

- метаевристики це стратегії, які керують процесом пошуку рішення.
- мета метаевристики полягає в ефективному досліджені простору пошуку для знаходження (майже) оптимальних рішень.
- метаевристичні алгоритми варіюють від простих процедур локального пошуку до складних процесів навчання.
- метаевристичні алгоритми є наближеними і, як правило, недетермінованими.

- метаевристичні алгоритми можуть включати механізми уникнення потрапляння в пастку в обмеженій області простору пошуку.
- метаевристики можуть бути описані на абстрактному рівні (тобто вони не призначені для вирішення конкретних завдань).
- метаевристики можуть використовувати предметно-орієнтовані знання у вигляді евристик, які знаходяться під контролем стратегії верхнього рівня.
- Сучасні метаевристики використовують збережений в пам'яті досвід пошуку рішення для управління пошуком.

Кожна метаевристика має свої власну поведінку і характеристики. Однак всі метаевристики мають ряд основних компонент і виконують операції в межах обмеженого числа категорій.

Ініціалізація. Метод знаходження початкового рішення.

Околиці. Кожному рішенню x відповідає безліч околиць і пов'язані з ними переходи: $\{N_1, N_2, \dots, N_q\}$.

Критерій вибору околиці визначається в разі наявності більше однієї околиці. Цей критерій повинен вказати не тільки обирану околицю, а й умови її вибору. Альтернативи варіюють від «на кожній ітерації» (наприклад, генетичні методи) до «за даних Умов».

Відбір кандидатів. Околиці можуть бути дуже великими. Тоді зазвичай розглядається тільки підмножина переходів на кожній ітерації. Відповідний список кандидатів $C(x) \subseteq N(x)$ може бути постійним і оновлюваним від ітерації до ітерації (наприклад, табу-пошук), або ж він може бути побудований на кожній новій ітерації (наприклад, генетичні методи). У всіх випадках критерій вибору визначає, яким чином можуть бути обрані рішення для включення в список кандидатів.

Критерій прийняття. Переходи оцінюються за допомогою функції $g(x, y)$, що залежить від таких параметрів двох рішень, як значення цільової функції, штрафи за порушення деяких обмежень і т. п. Вибирається найкраще рішення по відношенню до цього критерію $x = \text{argopt}\{g(x, y); y \in C(x)\}$ (з урахуванням необхідності запобігання зациклення).

Критерії зупинки. Метаевристика може бути зупинена згідно з різними критеріями: час обчислень, число ітерацій, темпи поліпшення рішення

4.1 Генетичні алгоритми

Основні принципи еволюційної теорії заклав Чарльз Дарвін у своїй найбільш революційній роботі - "Походження видів". Найважливішим його висновком був висновок про основну направлячу силу еволюції - нею визнавався **природний добір**. Іншими словами - виживає найсильніший (в широкому сенсі цього слова).

Другим, не менш важливим висновком Дарвіна був висновок про **мінливість** організмів.

4.1.1 Еволюційна теорія

Еволюційна теорія стверджує, що життя на нашій планеті виникло спочатку лише в найпростіших її формах — у вигляді одноклітинних організмів. Ці форми поступово ускладнювалися, пристосовуючись до навколошнього середовища і породжуючи нові види, і тільки через багато мільйонів років з'явилися перші тварини і люди. Можна сказати, що кожен біологічний вид з плином часу покращує свої якості так, щоб найбільш ефективно справлятися із завданнями виживання, самозахисту, розмноження і т. д.

Таким шляхом виникла захисне забарвлення у багатьох риб і комах, панцир у черепахи, отрута у Скорпіона і багато інших корисних пристосувань.

За допомогою еволюції природа постійно оптимізує все живе, знаходячи часом самі неординарні рішення. Дати цьому наукове пояснення можна, ґрунтуючись всього на двох біологічних механізмах:

- природного відбору;
- генетичного спадкування.

4.1.2 Природний відбір і генетичне спадкування

Ключову роль в еволюційній теорії відіграє **природний відбір**. Його суть полягає в тому, що найбільш пристосовані особини краще виживають і приносять більше потомства, ніж менш пристосовані.

Сам по собі природний відбір ще не забезпечує розвитку біологічного виду. Дійсно, якщо припустити, що всі нащадки народжуються приблизно однаковими, то різні покоління будуть відрізнятися тільки за чисельністю, але не по пристосованості. Тому дуже важливо вивчити, яким чином відбувається спадкування, тобто як властивості нащадка залежать від властивостей батьків.

Основний закон спадкування інтуїтивно зрозумілий кожному - він полягає в тому, що нащадки схожі на батьків. Зокрема, нащадки більш пристосованих батьків будуть, швидше за все, одними з найбільш пристосованих у своєму поколінні.

4.1.3 Задачі оптимізації

Як вже було зазначено вище, еволюція - це процес постійної оптимізації біологічних видів. Тепер ми в змозі зрозуміти, як відбувається цей процес. Природний відбір гарантує, що найбільш пристосовані особини дадуть досить велике потомство, а завдяки генетичному спадкуванню ми можемо бути впевнені, що частина цього потомства не тільки збереже високу пристосованість батьків, а й буде володіти і деякими новими властивостями. Якщо ці нові властивості виявляться корисними, то з великою ймовірністю вони перейдуть і в наступне покоління. Таким чином, відбувається накопичення корисних якостей і поступове підвищення пристосованості біологічного виду в цілому. Знаючи, як вирішується завдання оптимізації видів в природі, ми тепер застосуємо схожий метод для вирішення різних реальних завдань.

Завдання оптимізації - найбільш поширений і важливий для практики клас задач.

Як правило, в задачі оптимізації ми можемо управляти декількома параметрами (позначимо їх значення через x_1, x_2, \dots, x_n , а нашою метою є максимізація (або мінімізація) деякої функції, $f(x_1, x_2, \dots, x_n)$, залежить від цих параметрів. Функція f називається цільовою функцією.

У разі, якщо цільова функція досить гладка і має тільки один локальний максимум (унімодальна), то оптимальне рішення можна отримати методом градієнтного спуску. Ідея цього методу полягає в тому, що оптимальне рішення отримується ітераціями. Береться випадкова початкова точка, а потім в циклі відбувається зрушення цієї точки на малий крок, причому крок робиться в тому напрямку, в якому цільова функція зростає швидше за все. Недоліком градієнтного алгоритму є занадто високі вимоги до функції - на практиці унімодальність зустрічається вкрай рідко, а для неправильної функції градієнтний метод часто призводить до неоптимальної відповіді. Аналогічні проблеми виникають і з застосуванням інших математичних методів. У багатьох важливих завданнях параметри можуть приймати лише певні значення, причому у всіх інших точках цільова функція не визначена. Звичайно, в цьому випадку не може бути й мови про її гладкості і потрібні принципово інші підходи.

Класичний приклад такого завдання, відомий як «Завдання комівояжера» (Traveling Salesman Problem, TSP), формулюється так: комівояжеру потрібно об'їхати кілька міст, побувавши в кожному один раз, і повернутися у вихідну точку. Потрібно знайти найкоротший маршрут.

Найпростіший спосіб знайти оптимальне рішення - перебрати всі можливі значення параметрів. При цьому не потрібно робити ніяких припущення про властивості цільової функції, а задати її можна просто за допомогою таблиці. Однак, щоб вирішити таким способом Завдання комівояжера хоча б для 20 міст, потрібно перебрати близько 10^{19} маршрутів, що абсолютно нереально ні для якого обчислювального центру. Таким чином, виникає необхідність в будь-якому новому методі оптимізації, придатному для практики.

У наступному розділі ми покажемо, яким чином можна застосувати механізми еволюційного процесу до такого роду завдань. Фактично ми організуємо штучну еволюцію в спеціально побудованому світі.

4.1.4 Робота генетичного алгоритму

Уявімо собі штучний світ, населений безліччю істот (особин), причому кожна істота - це деяке рішення нашої задачі.

Будемо вважати особину тим більш пристосованою, чим краще відповідне рішення (чим більше значення цільової функції воно дає). Тоді завдання максимізації цільової функції зводиться до пошуку найбільш пристосованої істоти.

Звичайно, ми не можемо поселити в наш віртуальний світ всі істоти відразу, так як їх дуже багато. Замість цього ми будемо розглядати багато поколінь, що змінюють один одного.

Тепер, якщо ми зуміємо ввести в дію природний відбір і генетичне спадкування, то отриманий світ буде підкорятися законам еволюції. Зауважимо, що, відповідно до нашого визначення пристосованості, метою цієї штучної еволюції буде якраз створення найкращих рішень.

Очевидно, еволюція - нескінченний процес, в ході якого пристосованість особин поступово підвищується. Примусово зупинивши цей процес через досить довгий час після його початку і вибравши найбільш пристосовану особину в поточному поколінні, ми отримаємо не абсолютно точну, але близьку до оптимальної відповідь. Така, коротко, ідея генетичного алгоритму. Перейдемо тепер до точних визначень і опишемо роботу генетичного алгоритму більш детально.

Для того щоб говорити про генетичне спадкування, потрібно забезпечити наші істоти хромосомами.

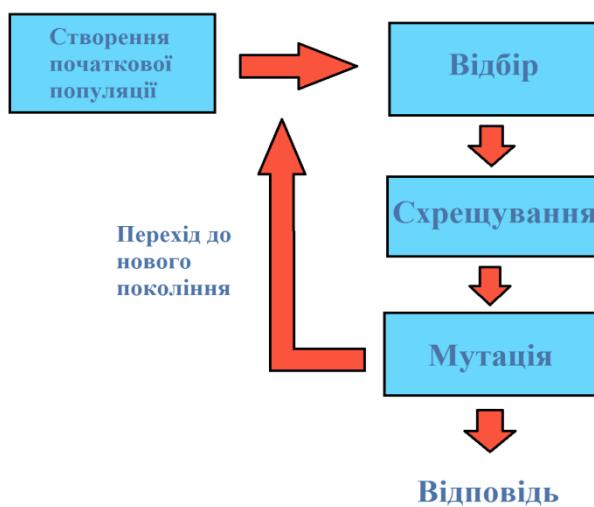
У генетичному алгоритмі **хромосома** - це деякий числовий вектор, відповідний параметру, що підбирається, а набір хромосом даної особини

визначає рішення задачі. Які саме вектори слід розглядати в конкретній задачі, вирішує сам користувач. Кожна з позицій вектора хромосоми називається геном.

Визначимо тепер поняття, відповідні мутації і кросинговеру в генетичному алгоритмі.

Мутація - це перетворення хромосоми, що випадково змінює одну або кілька її позицій (генів). Найбільш поширений вид мутацій-випадкова зміна тільки одного з генів хромосоми.

Кросинговер (в літературі за генетичними алгоритмами також вживається назва **кросовер** або **схрещування**) — це операція, при якій з двох хромосом породжується одна або кілька нових хромосом. У найпростішому випадку кросинговер в генетичному алгоритмі реалізується так само, як і в біології. При цьому хромосоми розрізаються у випадковій точці і обмінюються частинами між собою. Наприклад, якщо хромосоми (1, 2, 3, 4, 5) і (0, 0, 0, 0, 0) розрізати між третім і четвертим генами і обміняти їх частини, то вийдуть нащадки (1, 2, 3, 0, 0) і (0, 0, 0, 4, 5).



Блок-схема генетичного алгоритму зображена на рисунку.

Спочатку генерується початкова популяція особин (індивідуумів), тобто деякий набір рішень задачі. Як правило, це робиться випадковим чином.

Потім ми повинні моделювати розмноження всередині цієї популяції. Для цього випадково відбираються кілька пар індивідуумів, проводиться

схрещування між хромосомами в кожній парі, а отримані нові хромосоми поміщаються в популяцію нового покоління. В генетичному алгоритмі зберігається основний принцип природного відбору — чим більш пристосований індивідуум (чим більше відповідне йому значення цільової функції), тим з більшою ймовірністю він буде брати участь у схрещуванні.

Тепер моделюються мутації - в декількох випадково обраних особинах нового покоління змінюються деякі гени.

Потім стара популяція частково або повністю знищується і ми переходимо до розгляду наступного покоління. Популяція наступного покоління в більшості реалізацій генетичних алгоритмів містить стільки ж особин, скільки початкова, але в силу відбору пристосованість в ній в середньому вище.

Тепер описані процеси відбору, схрещування і мутації повторюються вже для цієї популяції і т. д.

У кожному наступному поколінні ми будемо спостерігати виникнення абсолютно нові рішення нашого завдання. Серед них будуть як погані, так і хороші, але, завдяки відбору, число хороших рішень буде зростати. Зауважимо, що в природі не буває абсолютнох гарантій, і навіть самий пристосований тигр може загинути від рушничного пострілу, не залишивши потомства. Імітуючи еволюцію на комп'ютері, ми можемо уникати подібних небажаних подій і завжди зберігати життя кращому з індивідуумів поточного покоління-така методика називається «стратегією елітизму».

4.1.5 Представлення генетичної інформації

Подібно до того, як хромосомний матеріал являє собою лінійну послідовність нуклеотидів одного з чотирьох типів, вектори змінних в ГА теж записуються у вигляді ланцюжка з використанням найчастіше двобуквеного алфавіту.

Будемо припускати, що кожна змінна закодована в певній ділянці хромосоми у вигляді гена. Хоча ми завжди говоримо про декодування, пряма

операція, яка розуміється як кодування, ніколи не застосовується. Хромосоми генерують випадковим чином шляхом послідовного заповнення рядів відразу в бінарному вигляді, і всякі наступні зміни в популяції зачіпають спочатку генетичний рівень, а тільки потім аналізуються фенотипічні цих змін.

В принципі, для декодування генетичної інформації з бінарної форми до десяткового виду підходить будь-який **двійково-десятковий код**, але зазвичай виходять з того, що він представлений кодом Грея, так як це дозволяє уникнути тупиків в порівнянні з додатковим кодом.

4.1.6 Мутація

Мутація зазвичай розглядається як «фоновий» процес, відповідальний за регенерацію ненавмисно «втрачених» значень генів, який запобігає збиванню популяції в одну купу.

Мутація є елементом рандомізованого пошуку в околиці поточного положення переважно популяції, що зійшлася.

На відміну від мутації, кросовер є основним процесом в ГА. проте, приклади з природи показують, що неповна репродукція може породити складні істоти без кросовера. Біологи розглядають мутацію як основне джерело еволюції. Шаффер провів великий експеримент в 89-му році з визначення оптимальних параметрів ГА. виявилося, що кросовер має набагато менший вплив на продуктивність, ніж думалося раніше. Було висунуто припущення про те, що «наївна еволюція» (тільки Селекція і мутація) являє собою пошук, схожий на метод градієнтного спуску і може бути досить ефективним без кросовера. Пізніше (в 91-му) ця гіпотеза була досліджена, в результаті чого було виявлено, що кросовер набагато прискорює еволюцію в порівнянні з тільки мутацією. У той же час, за допомогою мутацій виходили кращі рішення, ніж без використання її (тільки кросовер).

З'ясувалося також, що в міру сходження мутація стає більш продуктивною, а кросовер-менш продуктивним.

Незважаючи на дуже маленьку ймовірність застосування, мутація - це дуже важливий оператор. Оптимальне значення його ймовірності найбільш критичне, ніж для кросовера.

Спірс, досліджуючи кросовер і мутацію в порівнянні, прийшов до висновку, що в кожному з операторів є важливі характеристики, не присутні в іншому операторі. Далі він припустив, що спеціально пристосований оператор мутації може зробити те, що робить кросовер.

У 91-му році Ешелман прийшов до висновку, що ключ до ефективного застосування алгоритму «тільки мутації» лежить у використанні параметрів, закодованих кодом Грея, що робить пошук більш стійким до т. зв. «скель Хеммінга».

4.1.7 Загальна схема алгоритму

Нехай маємо комбінаторну задачу оптимізації:

$$\min\{f(S), S \in Sol\}.$$

Початкова популяція $P = \{S_1, S_2, \dots, S_k\}$ — набір допустимих рішень вихідної задачі.

Крок еволюції: вибираємо з популяції два рішення, схрещуємо їх, застосовуємо мутацію, локальне поліпшення і додаємо в популяцію, потім найгірше рішення видаляємо з популяції.

Тоді загальна схема генетичного алгоритму така:

1. Вибрати початкову популяцію P і запам'ятати рекорд $F^* = \min_{i=1,\dots,k} f(S_i)$.
2. Поки не виконано критерій зупинки виконувати наступне:
 - 2.1. Вибрати «батьків» S_{i_1}, S_{i_2} з популяції.
 - 2.2. Застосувати до S_{i_1}, S_{i_2} оператор схрещування і отримати нове рішення S' .
 - 2.3. З вірогідністю p застосувати до S' оператор мутації і отримати нове рішення S'' .
 - 2.4. Застосувати до S'' (чи S' , якщо не було мутації) оператор локального покращення и отримати нове рішення S''' .

- 2.5. Якщо $f(S'') < F^*$, то змінити рекорд $F^* := f(S'')$.
- 2.6. Додати S'' до популяції і видалити з неї найгірше рішення.

Вибір батьків

Турнірна селекція: з популяції P випадковим чином вибирається деяка підмножина $P' \subseteq P$ і батьком призначається найкраще рішення в P' :

$$S_i = \min_{S \in P'} f(S).$$

Пропорційна селекція: з популяції P випадковим чином вибираються два батьки. Для вирішення S_i імовірність бути обраним обернено пропорційна значенню цільової функції $f(S_i)$.

Варіанти: Кращий в популяції + випадково обраний.

Випадково обраний + найбільш віддалений від нього та ін.

Оператор скрещування

Нехай S_1, S_2 — два рішення, що задаються векторами $X^1, X^2 \in B^n$.

Одноточковий оператор скрещування: вибираємо випадковим чином координату $1 \leq l \leq n$ і новий вектор X' отримує перші l координат від вектора X^1 , а решта від вектора X^2 .

$$\begin{aligned} X^1: & (0 \ 1 \ 0 \ 0 \ 1 \ . \ . \ . \ 0 \ 1 \ 1) \\ X^2: & (1 \ 1 \ 0 \ 1 \ 0 \ . \ . \ . \ 1 \ 1 \ 1) \\ X': & (0 \ 1 \ 0 \ 0 \ 0 \ . \ . \ . \ 1 \ 1 \ 1) \end{aligned}$$

Аналогічно визначається двоточковий, триточковий і т. д. оператори.

Рівномірний оператор скрещування: нове рішення X' у кожній координаті отримує з ймовірністю 0.5 значення одного з батьків.

Оператор мутації

Ймовірнісний оператор мутації випадковим чином вносить зміни в

допустиме рішення задачі. Наприклад, з малою ймовірністю $q < \frac{1}{n}$ у кожній координаті значення $X_i \in \{0, 1\}$ замінюється на протилежне $1 - X_i$. Якщо в рішенні потрібно зберегти $\sum_{i \in I} x_i = b$, то випадковим чином вибирається координата i_1 така, що $X_{i_1} = 1$ і координата i_2 така, що $X_{i_2} = 0$ і проводиться заміна $X_{i_1} := 0$, $X_{i_2} := 1$.

$$X' = (0 \ 1 \ 0 \ 1 \ 0 \ . \ . \ . \ 0 \ 1 \ 1)$$

$$X'' = (1 \ 1 \ 0 \ 0 \ 0 \ . \ . \ . \ 0 \ 1 \ 1)$$

$i_1 \quad i_2$

Локальне покращення

Для рішення S позначимо через $N(S)$ його околицю, наприклад, безліч всіх рішень S' , що находяться від S на відстані не більше 2 (3, 4, 5)

Генетичні алгоритми мають наступні переваги:

1. Дозволяють вирішувати задачу будь-якої розмірності.
2. Дозволяють розпаралелити завдання.
3. Допускають обмеження рішення задачі, як за часом, так і за заданим значенням критерію.
4. Володіють високою швидкодією і можуть використовувати рішення, отримані іншими методами, як початкові наближення з їх можливою подальшою оптимізацією.
5. Мають велику кількість налаштувань, що дозволяють змінювати умови і швидкість збіжності.

Приклад

Задача про рюкзак - це одна з класичних задач дискретної оптимізації. Нехай є безліч предметів, кожен з яких має певну вартість і вагу. Потрібно скласти такий набір цих предметів, який мав би сумарну вартість, максимально можливу серед всіх наборів, чия сумарна вага не перевершує заданої величини – місткості рюкзака. Більш точно, нехай $c_i > 0$ і $a_i > 0$ – відповідно вартість і вага i -го предмета, де $i = 1, 2, 3, \dots, n$, а n – число предметів.

Потрібно знайти такий булевий вектор (x_1, x_2, \dots, x_n) , щоб була максимальна сума

$$\sum_{i=1}^n c_i x_i,$$

і виконувалась нерівність

$$\sum_{i=1}^n a_i x_i \leq P,$$

де $P > 0$ – місткість рюкзака.

До цього завдання часто зводяться практичні проблеми, що стосуються пошуку оптимального розподілу деякого ресурсу при наявності ряду обмежуючих факторів. Завдання про рюкзак, як і багато оптимізаційних комбінаторних завдань, належить до класу NP-повних завдань. Його можна вирішити повним перебором всіх допустимих варіантів заповнення рюкзака наявними предметами, однак при великих масивах входних даних такий перебірний алгоритм практично неприйнятний, оскільки має експоненціальну складність щодо довжини входу.

Нехай є 6 предметів, ваги і вартості яких вказані в таблиці. Будемо вважати, що місткість рюкзака $P = 8$.

Номер предмета	1	2	3	4	5	6
Стоимость	5	7	8	6	4	1
Вес	2	3	4	3	2	1

Початкова популяція $S_1=\{1,0,0,0,0,1\}=6$, $S_2=\{1,0,0,0,1,1\}=10$,
 $S_3=\{0,0,0,1,1,1\}=11$, $S_4=\{0,1,1,0,0,0\}=15$

$F^*=S_1$

Обираємо кращий в популяції + випадково вибраний.

$S_4=\{0,1,1,0,0,0\}=15 + S_1=\{1,0,0,0,0,1\}=6$

Оператор схрещування (одноточковий 3 і 3).

$S'=\{0,1,1,0,0,1\}=16$, $P=8$

Мутація: $S''=\{0,1,1,0,\textcolor{yellow}{1},1\}=20$, $\textcolor{red}{P=9}$

Невдала мутація.

$F^*=S'$

Новая популяція $S_1=\{0,1,1,0,0,1\}=16$, $S_2=\{1,0,0,0,1,1\}=10$,
 $S_3=\{0,0,0,1,1,1\}=11$, $S_4=\{0,1,1,0,0,0\}=15$

4.2 Мурашині алгоритми

Мурашині алгоритми являють собою имовірнісну жадібну евристику, де ймовірності встановлюються, виходячи з інформації про якість рішення, отриманої з попередніх рішень.

Вони можуть використовуватися як для статичних, так і для динамічних комбінаторних оптимізаційних завдань. Збіжність гарантована, тобто в будь-якому випадку ми отримаємо оптимальне рішення, однак швидкість збіжності невідома.

Почалося все з вивчення поведінки реальних мурах. Експерименти з Argentine ants, проведені Госсом в 1989 і Денеборгом в 1990 році послужили відправною точкою для подальшого дослідження ройового інтелекту. Дослідження застосування отриманих знань для дискретної математики почалися на початку 90-х років ХХ століття, автором ідеї є Марко Доріго з Університету Брюсселя, Бельгія. Саме він вперше зумів формалізувати поведінку мурах і застосувати стратегію їх поведінки для вирішення завдання

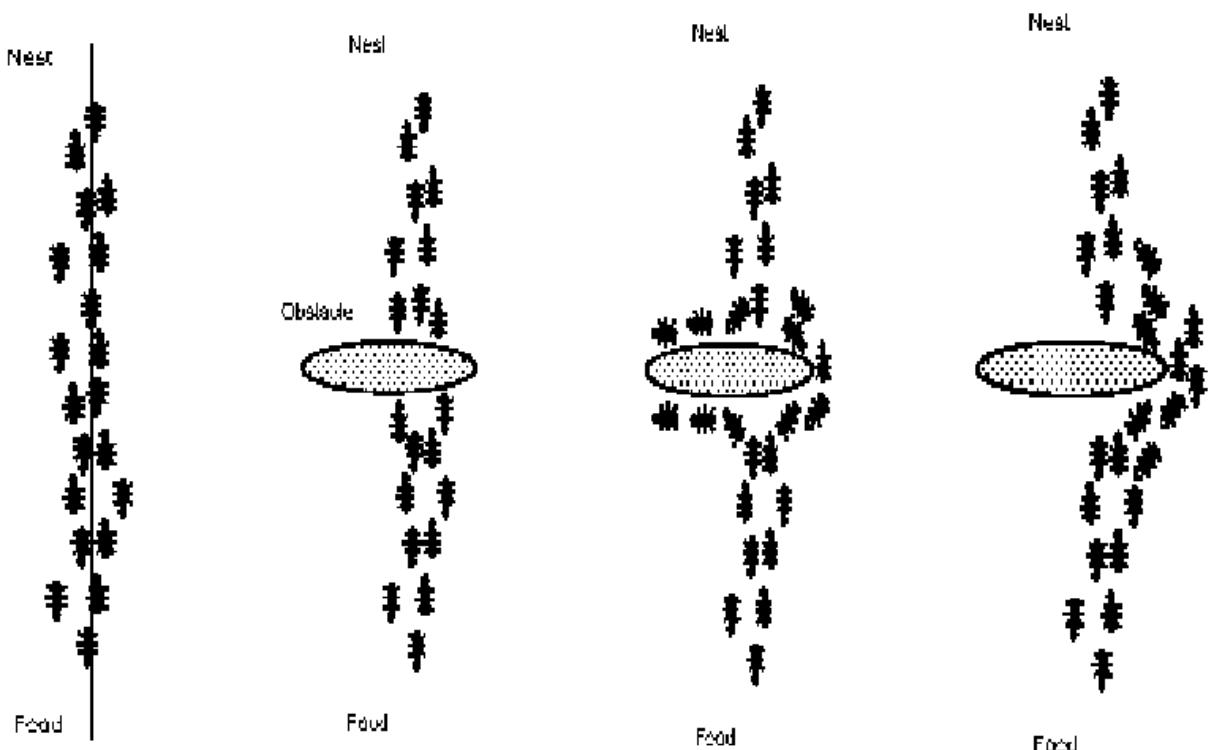
про найкоротших шляхах. Пізніше за участю Гамбарделли, Тайлларда і Ді Каро були розроблені і багато інших підходів до вирішення складних оптимізаційних завдань за допомогою муршиних алгоритмів. На сьогоднішній день ці методи є досить конкурентоспроможними в порівнянні з іншими евристиками і для деяких завдань дають найкращі на сьогоднішній день результати.

4.2.1 Концепція муршиних алгоритмів

Ідея муршиного алгоритму-моделювання поведінки мурах, пов'язаного з їх здатністю швидко знаходити найкоротший шлях від мурашника до джерела їжі і адаптуватися до мінливих умов, знаходячи новий найкоротший шлях.

При своєму русі мураха мітить шлях феромоном, і ця інформація використовується іншими мурахами для вибору шляху. Це елементарне правило поведінки і визначає здатність мурах знаходити новий шлях, якщо старий виявляється недоступним.

Розглянемо випадок, показаний на малюнку, коли на оптимальному досі шляху виникає перешкода. В цьому випадку необхідно визначення нового оптимального шляху. Дійшовши до перешкоди, мурахи з рівною ймовірністю будуть обходити її справа і зліва. Те ж саме буде відбуватися і на зворотному боці перешкоди. Однак, ті мурахи, які випадково виберуть найкоротший шлях, будуть швидше його проходити (туди/назад), і за кілька пересувань він буде більш збагачений феромоном. Оскільки рух мурах визначається **концентрацією феромону**, то наступні будуть віддавати перевагу саме цьому шляху, продовжуючи збагачувати його феромоном до тих пір, поки цей шлях з якої-небудь причини не стане недоступним.



Очевидний позитивний зворотний зв'язок швидко приведе до того, що найкоротший шлях стане єдиним маршрутом руху більшості мурах.

Моделювання випаровування феромону – **негативного зворотного зв'язку**-гарантуює нам, що **знайдене локально оптимальне рішення не буде єдиним**-мурахи будуть шукати і інші шляхи. Якщо ми моделюємо процес такої поведінки на деякому графі, ребра якого являють собою можливі шляхи переміщення мурах, протягом певного часу, то найбільш збагачений феромоном шлях по ребрах цього графа і буде рішенням завдання, отриманим за допомогою мурашиного алгоритму.

4.2.2 Узагальнений алгоритм

Будь-який мурашиний алгоритм, незалежно від модифікацій, представимо в наступному вигляді

Поки (умови виходу не виконані)

1. Створюємо мурах
2. Шукаємо рішення
3. Оновлюємо феромон
4. Додаткові дії {опціонально }

Тепер розглянемо кожен крок в циклі більш детально

Створюємо мурах

– **Стартова точка, куди поміщається мураха, залежить від обмежень, що накладаються умовами завдання.** Тому що для кожного завдання спосіб розміщення мурах є визначальним. Або всі вони поміщаються в одну точку, або в різні з повтореннями, або без повторень.

– На цьому ж етапі задається **початковий рівень феромона**. Він ініціалізується невеликим позитивним числом для того, щоб на початковому кроці ймовірності переходу в наступну вершину не були нульовими.

Шукаємо рішення

– Імовірність переходу з вершини i в вершину j в момент часу t визначається за такою формулою

$$p_{ij}(t) = \frac{\tau_{ij}(t)^\alpha \left(\frac{1}{d_{ij}}\right)^\beta}{\sum_{j \in J_i} \tau_{ij}(t)^\alpha \left(\frac{1}{d_{ij}}\right)^\beta},$$

де J_i – множина вершин, в які дозволений перехід з вершини i ,

$\tau_{ij}(t)$ – рівень феромона на дузі $i - j$ на кроці t ,

d_{ij} – відстань між вершинами i і j ,

α, β – константні параметри.

При $\alpha = 0$ вибір найближчого міста найбільш вірогідний, тобто алгоритм стає **жадібним**.

При $\beta=0$ вибір відбувається тільки на підставі феромона, що **призводить до локальних оптимумів** (субоптимальних рішень). Тому необхідний компроміс між цими величинами, який знаходиться експериментально.

Оновлюємо феромон

- Рівень феромону оновлюється відповідно до формули:

$$\tau_{ij}(t+1) = (1 - \rho)\tau_{ij}(t) + \sum_{k \in M_{ij}} \frac{L_{\min}}{L_k}$$

де ρ – інтенсивність випаровування,

M_{ij} – множина мурах, що пройшли по дузі (i, j) ,

L_k – ціна поточного рішення (довжина маршруту) для k -ї мурахи,

L_{\min} – параметр, що має значення порядку ціни оптимального рішення (передбачувана **ідеальна ціна рішення в задачі на мінімум**),

$\frac{L_{\min}}{L_k}$ – феромон, що відкладається k -ою мурахою на ребрі (i, j) . Таким

чином, чим гірше рішення (чим більше відповідне L_k), до якого увійшло ребро (i, j) , тим менше феромону буде відкладено мурахою на цьому ребрі.

Додаткові дії

Зазвичай тут використовується алгоритм локального пошуку, проте він може бути використаний і після пошуку всіх рішень.

4.2.3 Етапи вирішення задачі за допомогою мурашиних алгоритмів

Для того щоб побудувати відповідний мурашиний алгоритм для вирішення будь-якої задачі, необхідно виконати наступні дії.

1. Уявити завдання у вигляді набору компонент (вершин) і переходів (ребер) або набором неорієнтованих зважених графів, на яких мурахи можуть будувати рішення.
2. Визначити евристику поведінки мурашки при побудові рішення (в основі її лежить формула визначення ймовірностей переходів).
3. Визначити значення сліду феромону.

4. Якщо можливо, то визначити процедуру ефективного локального пошуку.

5. підібрати параметри АСО–алгоритму (α , β , ρ , L_{\min}).

Також визначальними є:

- кількість мурах;
- момент, коли оновлюється феромон;
- поєднання з жадібними евристиками або локальним пошуком.

4.2.4 Застосування мурашиних алгоритмів для задачі комівояжера

Завдання формулюється як задача пошуку мінімального за вартістю циклу (замкнутого маршруту) по всіх вершинах без повторень на повному зваженому графі з n вершинами. Змістово вершини графа є містами, які повинен відвідати комівояжер, а ваги ребер відображають відстані (довжини) або вартості проїзду. Це завдання є NP-важким, і точний переборний алгоритм його рішення має факторіальну складність.

Моделювання поведінки мурах пов'язано з розподілом феромону на стежці - дузі графа в задачі комівояжера. При цьому ймовірність включення ребра в цикл окремого мурашки пропорційна кількості феромону на цьому ребрі, а кількість феромона, що відкладали пропорційна довжині циклу. Чим коротше цикл, тим більше феромона буде відкладено на його ребрах, отже, більшу кількість мурах буде включати його в синтез власних циклів. Моделювання такого підходу, що використовує тільки позитивний зворотний зв'язок, призводить до передчасної збіжності - більшість мурашок рухається по локально оптимальному циклу. Уникнути цього можна, моделюючи **негативний зворотний зв'язок** у вигляді випаровування феромону. При цьому якщо феромон випаровується швидко, то це призводить до втрати пам'яті колонії і забування хороших рішень, з іншого боку, великий час випаровування може привести до отримання стійкого локального оптимального рішення.

4.2.5 Локальні правила поведінки мурах

Тепер з урахуванням особливостей задачі комівояжера, ми можемо описати локальні правила поведінки мурах при виборі шляху.

1. Мурахи мають власну «пам'ять». Оскільки кожне місто може бути відвідане тільки один раз, то у кожного мурашки є список вже відвіданих міст - **список заборон**. позначимо через J_{ik} – список міст, які необхідно відвідати мурасі k , що знаходиться в місті i .

2. Мурахи володіють «зором». Видимість - є евристичне бажання відвідати місто j , якщо мураха знаходиться в місті i . Будемо вважати, що видимість обернено пропорційна відстані між містами:

$$\eta_{ij} = \frac{1}{d_{ij}}.$$

3. Мурахи володіють «нюхом»-вони можуть вловлювати слід феромона, що підтверджує бажання відвідати місто j з міста i на підставі досвіду інших мурах. Кількість феромону на ребрі (i, j) в момент часу t позначимо через $\tau(i, j)$

4. На цій підставі ми можемо сформулювати ймовірнісно-пропорційне правило, що визначає ймовірність переходу k -ої мурахи з міста i в місто j :

$$p_{ij}(t) = \frac{\tau_{ij}(t)^\alpha (\eta_{ij})^\beta}{\sum_{j \in J_i} \tau_{ij}(t)^\alpha (\eta_{ij})^\beta}, \quad j \in J_{ik}, \\ p_{ij}(t) = 0, \quad j \notin J_{ik}, \quad (1)$$

де α, β – параметри, що задають ваги сліду феромону. При $\alpha = 0$ алгоритм вироджується до жадного алгоритму (буде обране найближче місто).

Зауважимо, що вибір міста є імовірнісним. Правило (1) не змінюється в ході алгоритму, але у двох різних мурах значення ймовірностей переходу будуть відрізнятися, тому що вони мають різний список J_{ik} дозволених міст.

5. Пройшовши ребро (i, j) , мураха відкладає на ньому деяку кількість феромону, яке повинна бути пов'язана з якістю обраного циклу. нехай $T_k(t)$ – цикл, пройдений мурахою k до моменту часу t , $L_k(t)$ – довжина цього циклу, а L_{\min} – параметр, що має значення порядку довжини оптимального циклу. Тоді кількість феромону, що відкладається мурахою k може бути задана у вигляді:

$$\Delta\tau_{ij,k} = \begin{cases} \frac{L_{\min}}{L_k}, & (i, j) \in T_k(t) \\ 0, & (i, j) \notin T_k(t) \end{cases}$$

Правила зовнішнього середовища визначають, в першу чергу, випаровування феромону. Нехай ρ – коефіцієнт випаровування, тоді правило випаровування має вигляд:

$$\begin{aligned} \tau_{ij}(t+1) &= (1 - \rho)\tau_{ij}(t) + \Delta\tau_{ij}(t), \\ \Delta\tau_{ij}(t) &= \sum_{k=1}^M \Delta\tau_{ij,k}(t), \end{aligned} \tag{2}$$

де M – кількість мурах в колонії

На початку алгоритму кількість феромону на ребрах приймається рівною невеликому позитивному числу. Загальна кількість мурах залишається постійною і рівною кількості міст, кожна мураха починає маршрут зі свого міста.

Додаткова модифікація алгоритму може складатися у веденні так званих «елітних» мурах, які підсилюють ребра найкращого маршруту, знайденого з початку роботи алгоритму. позначимо через T^* – найкращий поточний цикл, а через L^* – його довжину. Тоді після визначення елітної мурашки ребра найкращого поточного циклу T^* отримають додаткову кількість феромону

$$\Delta\tau = \frac{L_{\min}}{L^*}. \tag{3}$$

Якщо ж в колонії є l елітних мурах, то аналогічно, ребра відповідних їм найкращих l циклів отримають додаткову кількість феромону.

4.2.6 Мурашиний алгоритм для задачі комівояжера в псевдокоді

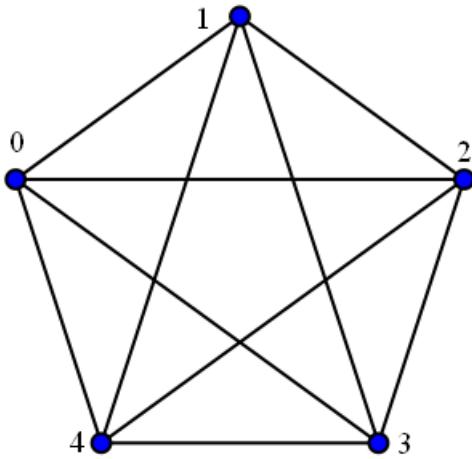
1. Введення матриці відстаней D .
2. Ініціалізація параметрів алгоритму – $\alpha, \beta, \rho, L_{\min}, l$.
3. Визначення $L_{\min}, L^* = \infty$.
4. Ініціалізація ребер – привласнення видимості η_{ij} і початкової концентрації феромона.
5. Рівномірне розміщення мурах в міста.
//Основний цикл
6. ЦИКЛ ЗА ЧАСОМ ЖИТТЯ КОЛОНІЇ $t = \overline{1, t_{\max}}$.
7. ЦИКЛ ПО УСІХ МУРАХАХ $k=1, m$
8. Побудувати цикл обходу міст $T_k(t)$ по правилу (1) і розрахувати його довжину $L_k(t)$.
9. КІНЕЦЬ ЦИКЛУ ПО МУРАХАХ
10. Порівняння всіх $L_k(t)$ і вибір серед них найменшого (відповідного найкращому поточному рішенню $T^*(t)$).
11. Перевизначити L^* и T^* .
12. ЦИКЛ ПО ВСІХ РЕБРАХ ГРАФА
13. Оновити сліди феромону на ребрі по правилам (2) і (3)
14. КІНЕЦЬ ЦИКЛУ ПО РЕБРАХ ГРАФА
15. КІНЕЦЬ ЦИКЛУ ЗА ЧАСОМ ЖИТТЯ
16. Вивести найкоротший цикл обходу міст T^* і його довжину L^* .

Модифікації:

Розміщення мурах без збігів в випадково вибрані міста.

Складність даного алгоритму залежить від часу життя колонії t_{\max} , кількості міст n і кількості мурах в колонії M

Приклад



$$D = \begin{pmatrix} \infty & 1 & 7 & 3 & 14 \\ 3 & \infty & 6 & 9 & 1 \\ 6 & 14 & \infty & 3 & 7 \\ 2 & 3 & 5 & \infty & 9 \\ 15 & 7 & 11 & 2 & \infty \end{pmatrix}.$$

$$\alpha = 2, \beta = 3, \rho = 0.2, L_{\min} = 34, L^* = \infty.$$

$M=1$ – кількість мурах

Запишемо матриці видимості і концентрації феромонів

$$d_{ij} := \begin{pmatrix} 0 & \frac{1}{1} & \frac{1}{7} & \frac{1}{3} & \frac{1}{14} \\ \frac{1}{2} & 0 & \frac{1}{6} & \frac{1}{9} & \frac{1}{1} \\ \frac{1}{6} & \frac{1}{14} & 0 & \frac{1}{3} & \frac{1}{7} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{5} & 0 & \frac{1}{9} \\ \frac{1}{15} & \frac{1}{7} & \frac{1}{11} & \frac{1}{2} & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0.143 & 0.333 & 0.071 \\ 0.5 & 0 & 0.167 & 0.111 & 1 \\ 0.167 & 0.071 & 0 & 0.333 & 0.143 \\ 0.5 & 0.333 & 0.2 & 0 & 0.111 \\ 0.067 & 0.143 & 0.091 & 0.5 & 0 \end{pmatrix}$$

$$\tau_{ij} := \begin{pmatrix} 0 & 0.2 & 0.1 & 0.1 & 0.2 \\ 0.1 & 0 & 0.3 & 0.2 & 0.1 \\ 0.3 & 0.3 & 0 & 0.3 & 0.3 \\ 0.3 & 0.2 & 0.3 & 0 & 0.3 \\ 0.2 & 0.3 & 0.1 & 0.3 & 0 \end{pmatrix}$$

Початкова вершина 0.

Знайдемо ймовірності переходу в вершини 1, 2, 3, 4 з вершини 0.

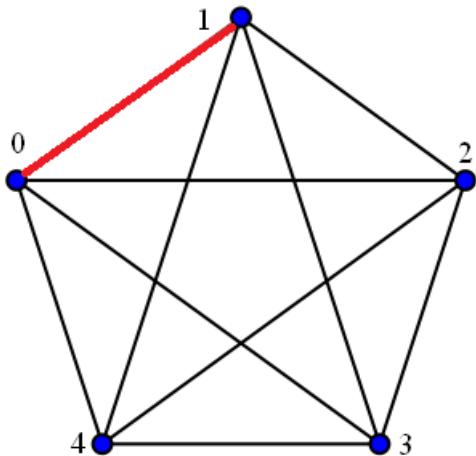
$$p_{01} := \frac{(\tau_{ij0,1})^2 \cdot (dij_{j0,1})^3}{\sum_{j=1}^4 [(\tau_{ij0,j})^2 \cdot (dij_{j0,j})^3]} = 0.99$$

$$p_{02} := \frac{(\tau_{ij0,2})^2 \cdot (dij_{j0,2})^3}{\sum_{j=1}^4 [(\tau_{ij0,j})^2 \cdot (dij_{j0,j})^3]} = 7.214 \times 10^{-4}$$

$$p_{03} := \frac{(\tau_{ij0,3})^2 \cdot (dij_{j0,3})^3}{\sum_{j=1}^4 [(\tau_{ij0,j})^2 \cdot (dij_{j0,j})^3]} = 9.164 \times 10^{-3}$$

$$p_{04} := \frac{(\tau_{ij0,4})^2 \cdot (dij_{j0,4})^3}{\sum_{j=1}^4 [(\tau_{ij0,j})^2 \cdot (dij_{j0,j})^3]} = 3.607 \times 10^{-4}$$

Ідемо в вершину 1.



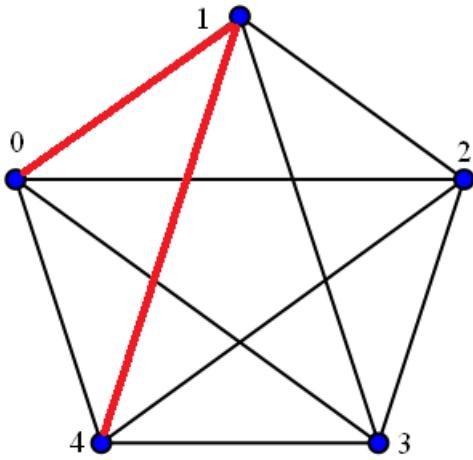
Знайдемо ймовірності переходу в вершини 2, 3, 4 з вершини 1.

$$p_{12} := \frac{(\tau_{ij1,2})^2 * (dij_{j1,2})^3}{\sum_{j=2}^4 [(\tau_{ij1,j})^2 * (dij_{j1,j})^3]} = 0.04$$

$$p_{13} := \frac{(\tau_{ij1,3})^2 * (dij_{j1,3})^3}{\sum_{j=2}^4 [(\tau_{ij1,j})^2 * (dij_{j1,j})^3]} = 0.005$$

$$p_{14} := \frac{(\tau_{ij1,4})^2 * (dij_{j1,4})^3}{\sum_{j=2}^4 [(\tau_{ij1,j})^2 * (dij_{j1,j})^3]} = 0.955$$

Йдемо в вершину 4.

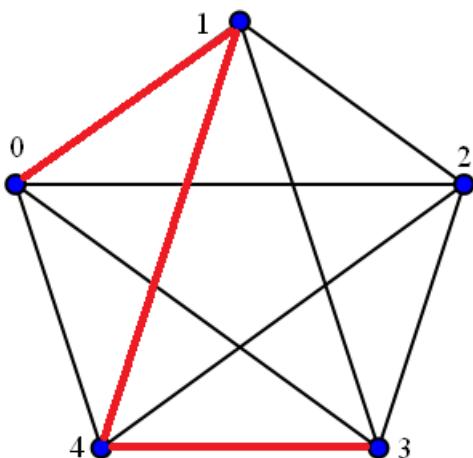


Знайдемо ймовірності переходу в вершини 2, 3 з вершини 4.

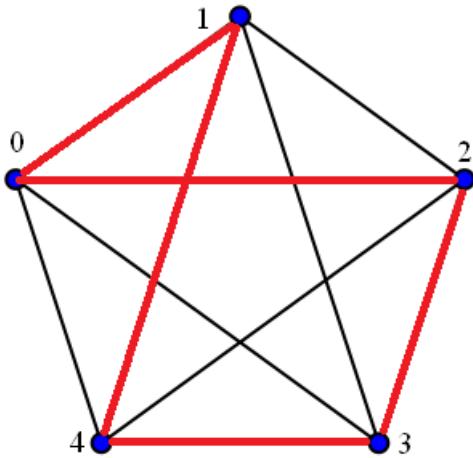
$$p_{42} := \frac{(\tau_{ij4,2})^2 \cdot (\text{dij}_4,2)^3}{(\tau_{ij4,2})^2 \cdot (\text{dij}_4,2)^3 + (\tau_{ij4,3})^2 \cdot (\text{dij}_4,3)^3} = 6.674 \times 10^{-4}$$

$$p_{43} := \frac{(\tau_{ij4,3})^2 \cdot (\text{dij}_4,3)^3}{(\tau_{ij4,2})^2 \cdot (\text{dij}_4,2)^3 + (\tau_{ij4,3})^2 \cdot (\text{dij}_4,3)^3} = 0.999$$

Йдемо в вершину 3.



Далі ми можемо піти тільки в вершину 2 і повернутися в 0.



Знайдемо довжину маршруту $L = 1+1+2+5+6=15$.

Перерахуємо концентрацію феромона.

$$\tau_{ij0,1} := (1 - 0.2) \cdot \tau_{ij0,1} + \frac{34}{15} = 2.427$$

$$\tau_{ij0,2} := (1 - 0.2) \cdot \tau_{ij0,2} = 0.08$$

$$\tau_{ij0,3} := (1 - 0.2) \cdot \tau_{ij0,3} = 0.08$$

$$\tau_{ij0,4} := (1 - 0.2) \cdot \tau_{ij0,4} = 0.16$$

$$\tau_{ij1,0} := (1 - 0.2) \cdot \tau_{ij1,0} = 0.08$$

$$\tau_{ij1,2} := (1 - 0.2) \cdot \tau_{ij1,2} = 0.24$$

$$\tau_{ij1,3} := (1 - 0.2) \cdot \tau_{ij1,3} = 0.16$$

$$\tau_{ij1,4} := (1 - 0.2) \cdot \tau_{ij1,4} + \frac{34}{15} = 2.347$$

$$\tau_{ij2,0} := (1 - 0.2) \cdot \tau_{ij2,0} + \frac{34}{15} = 2.507$$

$$\tau_{ij2,1} := (1 - 0.2) \cdot \tau_{ij2,1} = 0.24$$

$$\tau_{ij2,3} := (1 - 0.2) \cdot \tau_{ij2,3} = 0.24$$

$$\tau_{ij2,4} := (1 - 0.2) \cdot \tau_{ij2,4} = 0.24$$

$$\begin{aligned}\tau_{ij3,0} &:= (1 - 0.2) \cdot \tau_{ij3,0} = 0.24 \\ \tau_{ij3,1} &:= (1 - 0.2) \cdot \tau_{ij3,1} = 0.16 \\ \tau_{ij3,2} &:= (1 - 0.2) \cdot \tau_{ij3,2} + \frac{34}{15} = 2.507 \\ \tau_{ij3,4} &:= (1 - 0.2) \cdot \tau_{ij3,4} = 0.24\end{aligned}$$

$$\begin{aligned}\tau_{ij4,0} &:= (1 - 0.2) \cdot \tau_{ij4,0} = 0.16 \\ \tau_{ij4,1} &:= (1 - 0.2) \cdot \tau_{ij4,1} = 0.24 \\ \tau_{ij4,2} &:= (1 - 0.2) \cdot \tau_{ij4,2} = 0.08 \\ \tau_{ij4,3} &:= (1 - 0.2) \cdot \tau_{ij4,3} + \frac{34}{15} = 2.507\end{aligned}$$

Запишемо матрицю концентрації феромонів

$$\tau_{ij} = \begin{pmatrix} 0 & 2.427 & 0.08 & 0.08 & 0.16 \\ 0.08 & 0 & 0.24 & 0.16 & 2.347 \\ 2.507 & 0.24 & 0 & 0.24 & 0.24 \\ 0.24 & 0.16 & 2.507 & 0 & 0.24 \\ 0.16 & 0.24 & 0.08 & 2.507 & 0 \end{pmatrix}$$

Кінець ітерації.

4.3 Алгоритм бджолиної колонії

Даний алгоритм моделює поведінку бджіл в природному середовищі. Запропоновано Д. Карабога в 2005 р.

Основною метою роботи бджолиної колонії в природі є розвідка простору навколо вулика з метою пошуку нектару з подальшим його збором. Для цього в складі колонії існують різні типи бджіл: бджоли-розвідники і робочі бджоли-фуражири (крім них, в колонії існують трутні і матка, що не беруть участі в процесі збору нектару). Розвідники ведуть дослідження навколошнього простору вулика і повідомляють інформацію про перспективні місця, в яких було виявлено найбільшу кількість нектару (для обміну інформацією в вулику існує спеціальний механізм, іменований танцем бджоли). Далі по найбільш перспективним напрямкам вилітають робочі бджоли, які займаються збором нектару, попутно проводячи уточнення інформації розвідників про кількість нектару в деякій околиці від зазначеного розвідником області. Робота зазначених

типів бджіл у вулику забезпечує ефективну розвідку навколоишнього простору і збір нектару.

Ідея бджолиного алгоритму полягає в тому, що всі бджоли на кожному кроці будуть вибирати як елітні ділянки для дослідження, так і ділянки в околиці елітних, що дозволить, по-перше, урізноманітнити популяцію рішень на наступних ітераціях, по-друге, збільшити ймовірність виявлення близьких до оптимальних рішень. Наведемо основні поняття бджолиного алгоритму:

1. Джерело нектару (квітка, ділянка).
2. Фуражири (робочі бджоли).
3. Бджоли-розвідники.

Джерело нектару характеризується значимістю, яка визначається різними параметрами. Фуражири закріплені за джерелами нектару. Кількість всіх бджіл у цих ділянках більше, ніж на інших. Середня кількість розвідників в рої становить 5 - 10%. Повернувшись у вулик, бджоли обмінюються інформацією за допомогою танців на так званому закритому майданчику для танців.

Нехай рішення являє собою вектор H . Областю пошуку нектару для бджіл буде простір пошуку рішень, розмірністю $n!$ (кількість всіх можливих перестановок вектора H). Розташування джерела нектару характеризується конкретною перестановкою H , рішенням. Таким чином, координатами джерела є рішення H . Кількість нектару на джерелі пропорційно ЦФ (в задачах максимізації) і обернено пропорційно ЦФ (в задачах мінімізації). Ділянка має розміри, де розмір - кількість рішень, «блізьких» до H . Близькість між векторами можна визначити значенням відстані Хеммінга між ними. Наприклад, рішення $\{5,2,7,3,4,1,6\}$ і $\{5,4,7,3,2,1,6\}$ є «блізькими», тобто в просторі пошуку вони розташовуються поруч, знаходяться на одному «ділянці».

Відстань Хеммінга – число позицій, в яких відповідні символи двох слів однакової довжини різні. У більш загальному випадку відстань Хеммінга застосовується для рядків однакової довжини будь-яких q -ічних алфавітів і служить метрикою відмінності (функцією, визначальною відстань в метричному просторі) об'єктів однакової розмірності.

Наведемо словесний опис алгоритму бджіл.

- 1) Генерація ділянок для пошуку нектару.
- 2) Оцінка корисності ділянок.
- 3) Вибір ділянок для пошуку в їх околиці.
- 4) Відправка фуражирів.
- 5) Пошук в околицях джерел нектару.
- 6) Оновити оцінки корисності ділянок.
- 7) Відправка бджіл-розвідників.
- 8) Випадковий пошук (опціонально).
- 9) Якщо умова зупинки не виконується, то п. 2.
- 10) Кінець роботи алгоритму.

Розфарбування графа

Граф G називають l - хроматичним, якщо його вершини можуть бути розфарбовані з використанням l кольорів (фарб) так, що не знайдеться двох суміжних вершин одного кольору. найменше число l таке, що граф G являється χ - хроматичним, називається хроматичним числом графа G і позначається $\chi(G)$. Завдання знаходження хроматичного числа графа називається завданням про розфарбовування графа. Відповідне цьому числа розфарбування вершин розбиває безліч вершин графа на l підмножин, кожне з яких містить вершини одного кольору. Ці множини є незалежними, оскільки в межах однієї множини немає двох суміжних вершин.

4.3.1 Алгоритм бджолиної колонії для розмальовки графа (класичний випадок).

Задан граф и степінь його вершин.

Задано загальне число бджіл і число розвідників.

Є палітра всіх доступних кольорів і палітра використовуваних.

1. Генерація ділянок для пошуку нектару.

1.1. Потрібно згенерувати безліч початкових рішень (мінімум 1), наприклад жадібним алгоритмом.

2. Оцінка корисності ділянок.

2.1. На ділянки відправляються бджоли розвідники (в загальному випадку початкову розвідку варто провести на всіх ділянках).

3. Вибір ділянок для пошуку в їх околиці.

3.1. Є 2 сценарії:

3.1.1. перевірити всі ділянки і вибрати найбільш перспективні (можливо локальне рішення);

3.1.2. вибирати кращий з декількох (по числу бджіл-розвідників) випадкових ділянок;

4. Відправка фуражирів.

4.1. Відправляються всі фуражири крім розвідників, але не більше ніж максимальний ступінь вершини в графі, інші можуть бути задіяні на інших ділянках.

5. Пошук в околицях джерел нектару.

5.1. Робимо заміну кольору вершини із суміжними і намагаємося зменшити число кольорів (шляхом підстановки з палітри використовуваних), повторюємо для кожної суміжної вершини.

5.2. Переходимо на наступну вершину з найбільшим ступенем (можна зробити чергу з пріоритетом), і повторюємо 5.1, поки вершини не закінчилися.

6. Оновити оцінки корисності ділянок.**7. Відправка бджіл-розвідників.**

7.1. Відправляємо задане число розвідників на ділянки.

8. Випадковий пошук (опціонально).

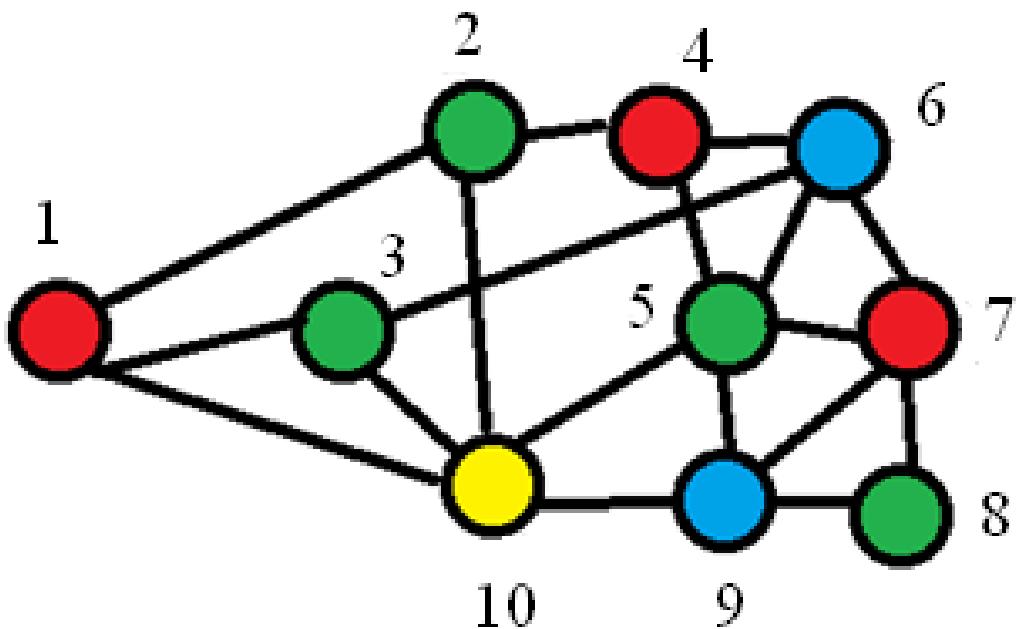
8.1. Наприклад, вибрati випадкову ділянку.

9. Якщо умова зупинки не виконується, то п. 2.**10. Кінець роботи алгоритму.**

Таким чином, ключовою операцією алгоритму бджіл є спільне дослідження перспективних областей і їх околиць. В кінці роботи алгоритму безліч рішень буде складатися з двох частин: ділянки з кращими значеннями цільової функції (елітні), а також групи ділянок з випадковими значеннями ЦФ. Залежність часової складності бджолиного алгоритму від числа вершин - $O(n^2)$.

Приклад

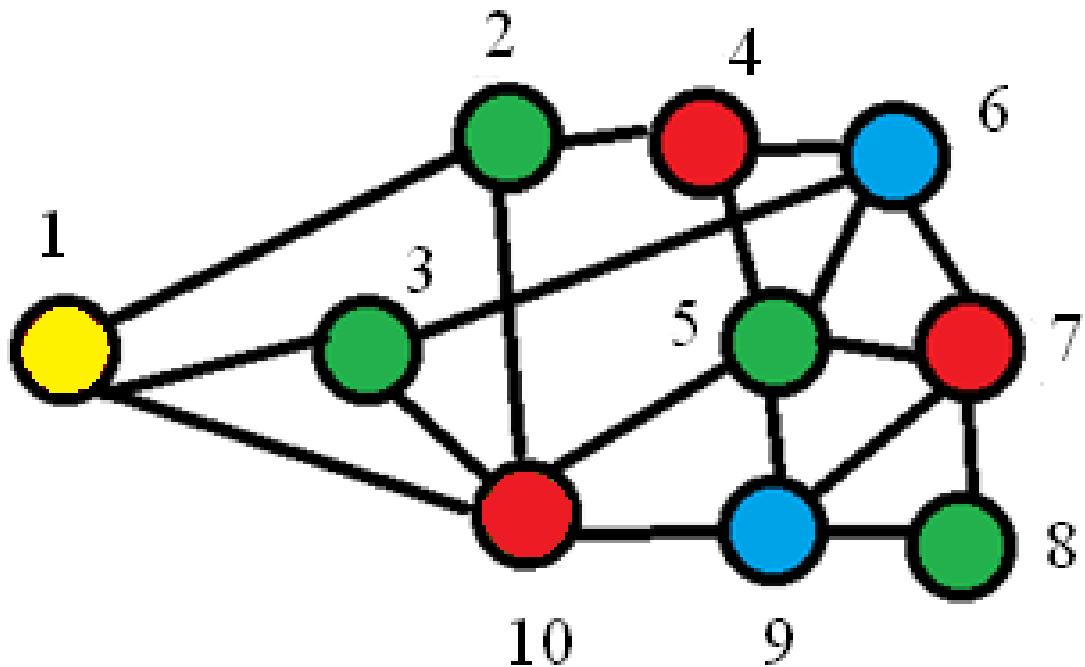
Розглянемо граф на 10 вершин. Для початку знайдемо допустиме рішення жадібним алгоритмом (генерація ділянок, в даному випадку 1). Для генерації безлічі ділянок варто використовувати різні шляхи розмальовки.



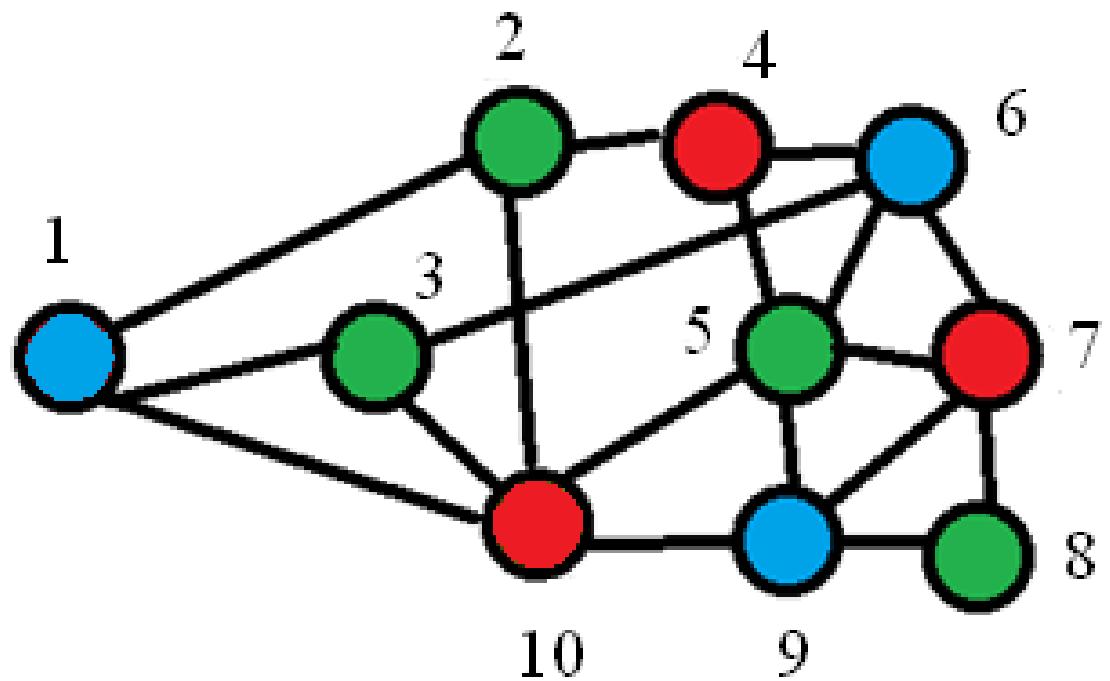
Хроматичне число графа = 4 (оцінка корисності).

Знайдемо вершину з максимальним ступенем і поміняємо її колір з кольором суміжної вершини (пошук в околицях джерел нектару).

Таку процедуру повторимо для всіх суміжних, кількість повторів визначається числом фуражирів. У загальному випадку вершину для заміни кольорів із суміжними вершинами можна вибирати випадково.



Якщо заміна допустима, то намагаємося зменшити число кольорів.



Хроматичне число графа = 3

4.3.2 ABC-алгоритм

Граф складається з багаточисленних вузлів і ребер.

Вузли графа розглядаються як джерела їжі в реальній моделі і степінь кожної вершини визначає кількість нектару.

Всі бджоли на початку діляться на зайнятих (розвідники) і спостерігачів (фуражири).

Алгоритм можна узагальнити наступним чином:

Попередній етап

Кожна зайнята бджола вибирає одну вершину графа випадковим чином, а потім переходить на неї (зауважимо, що для кожного вузла призначена тільки одна зайнята бджола).

Далі всі зайняті бджоли обчислюють степінь своєї вершини (кількість нектару), щоб поділитися ним зі спостерігачами у вулика.

Вага нектару використовується пізніше бджолами-спостерігачами для того, щоб перейти на позицію, яка визначається зайнятими бджолами наступним чином:

$$p_i = \frac{\text{nectar}_i}{\sum_{i=1}^{NB} \text{nectar}_i}$$

де NB - кількість бджіл (зайнятих), nectar_i - степінь вершини i .

Кількість зайнятих бджіл не рекомендується брати більше половини від числа вершин входного графа.

Основний алгоритм

1. Після того, як зайняті бджоли повертаються у вулик, вони діляться інформацією про кількість вузлів з бджолами-спостерігачами.
2. Спостерігачі переміщаються в вершину, яка визначається зайнятою бджолою за допомогою рівняння для p_i .

Якщо ступінь цієї вершини дорівнює нулю, потрібно їй присвоїти перший колір з доступних і у нас не буде жодних обмежень у використанні цього кольору для інших вершин, оскільки він не суміжний з жодною вершиною в графі.

3. Знайдемо суміжні вершини до поточної і дамо їм випадковий доступний колір з можливих (намагаємося обійтися мінімальним числом кольорів). Можуть бути варіанти 4-6.

4. Присвоєні кольори порівнюються з сусідніми вершинами, і якщо вони будуть допустимі, вони будуть додані до масиву Used Colors, що містить кольори, які використовуються на даний момент.

5. В іншому випадку, якщо вони не допустимі, приймаються інші кольори, з уже використаних кольорів, які зберігаються в масиві Used Colors.

6. В іншому випадку, якщо спостерігачі не можуть знайти відповідного кольору в масиві Used Colors, вони будуть вибирати новий колір з масиву All Colors. Потім обраний колір додається до пам'яті (масив Used Colors).

Число пофарбованих суміжних вершин дорівнює числу спостерігачів відправлених в обрану.

7. Після розфарбування всіх сусідів обраної вершини (нектар поточної вершини віднімається тільки коли кожна з її сусідів забарвлена), бджола-спостерігач повинна вибрати колір поточної вершини. Цей колір вибирається з масиву Used Colors або All Colors.

8. Бджола-спостерігач перетворюється в розвідника і йде до випадкової з не відвіданих вершини, після розмальовки поточної вершини (встановлюється нектар поточної вершини нулем, коли спостерігач змінюється на розвідника).

Повторити п. 3-8 для всіх розвідників (зайнятих бджіл), а далі повторити п. 1-10, поки всі вершини не будуть пофарбовані.

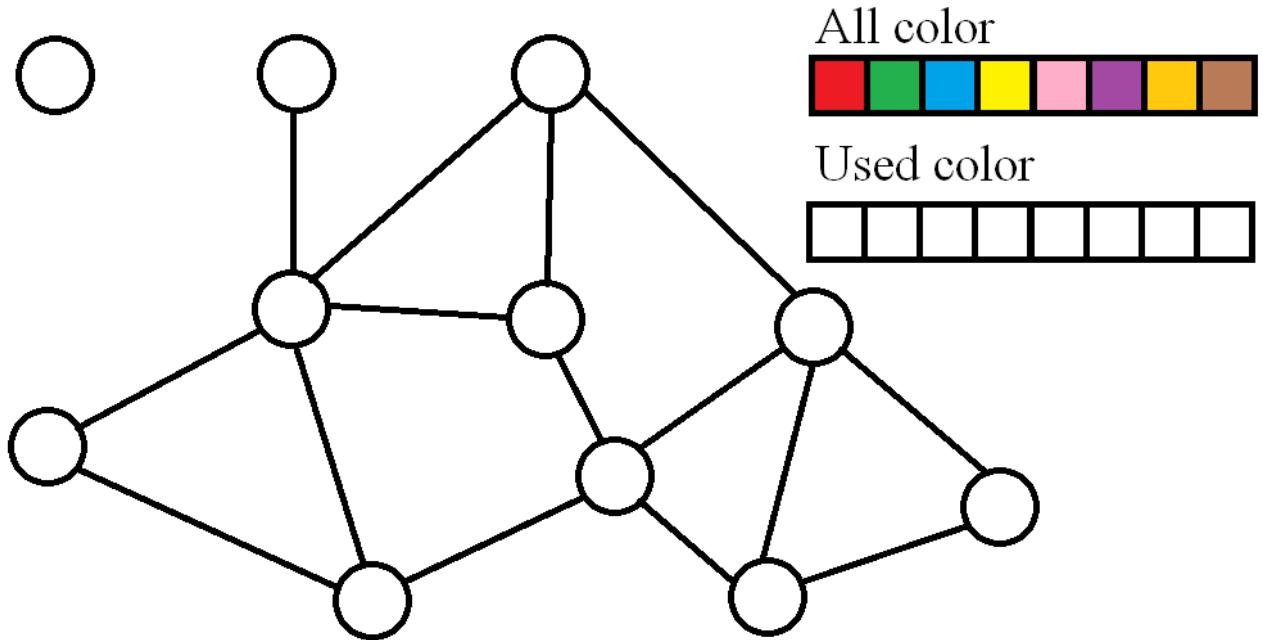
Зафіксувати хроматичне число.

Повторювати процедуру поки не буде досягнуто потрібне хроматичне число або задане число ітерацій.

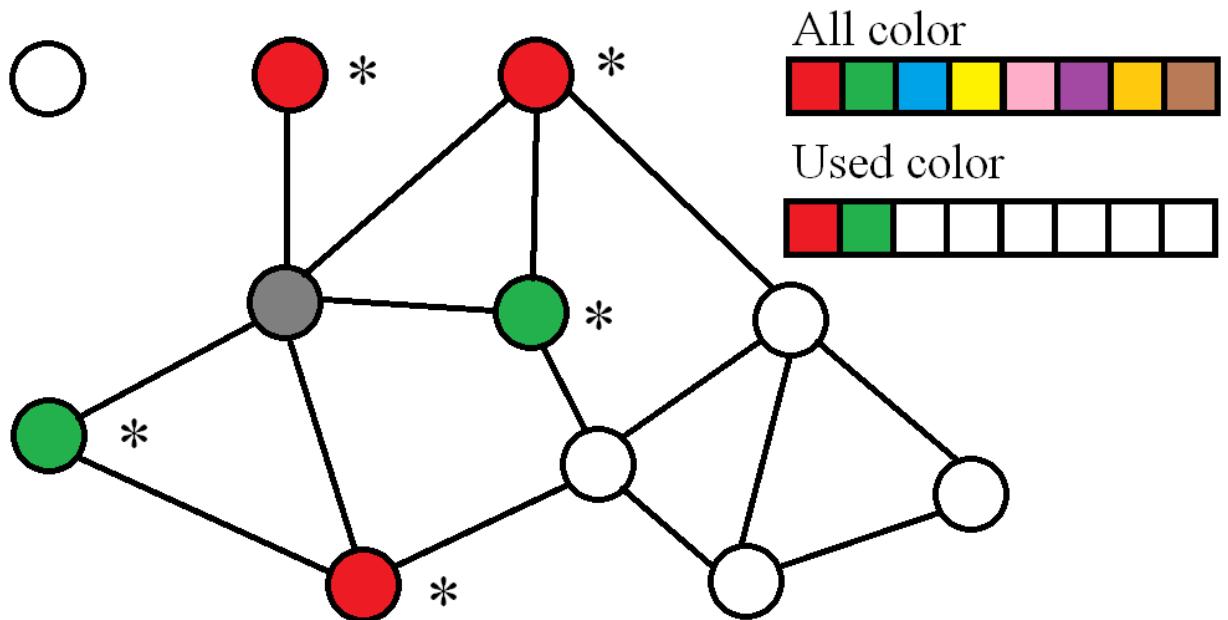
Якщо на наступних ітераціях значення хроматичного числа стало гірше, ми залишаємо колишнє значення.

Приклад

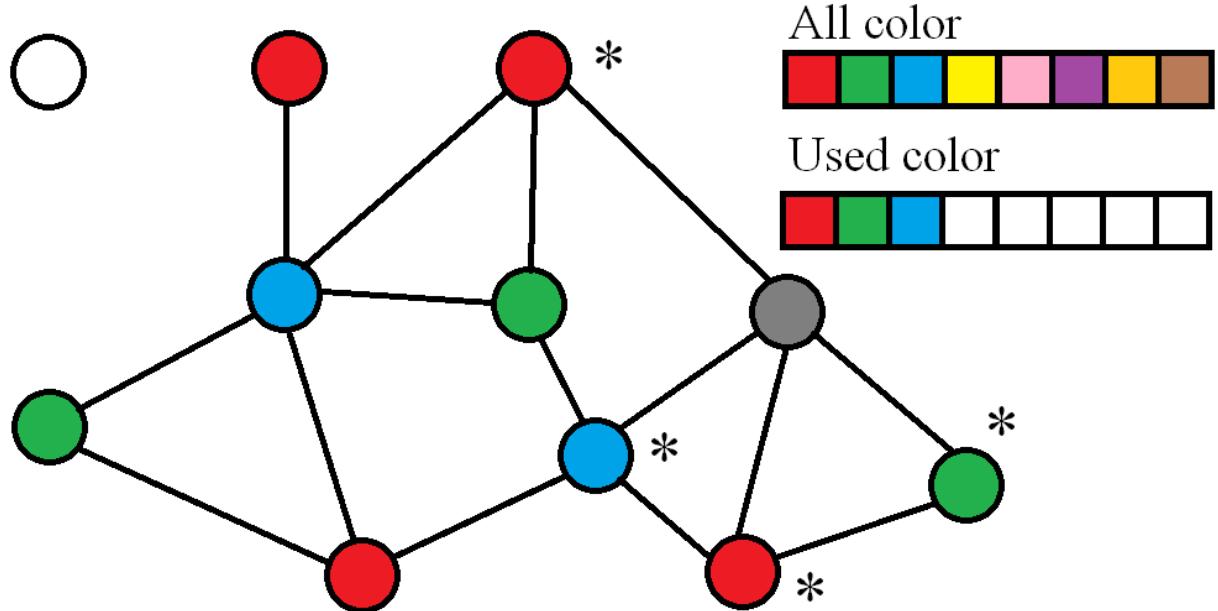
Заданий граф на 11 вершин, число зайнятих бджіл (розвідників) - 1, число спостерігачів (фуражирів) - 5.



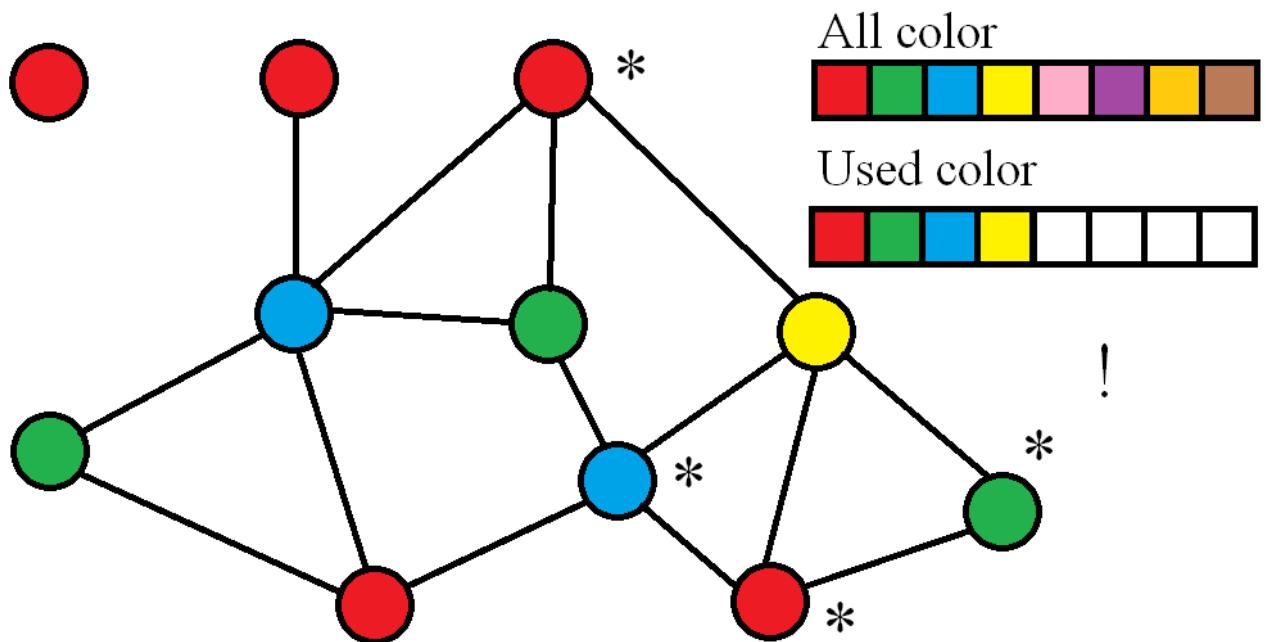
Вибираємо одну (1 розвідник) з вершин і фарбуємо суміжні з нею (5 максимум тому 5 фуражирів).



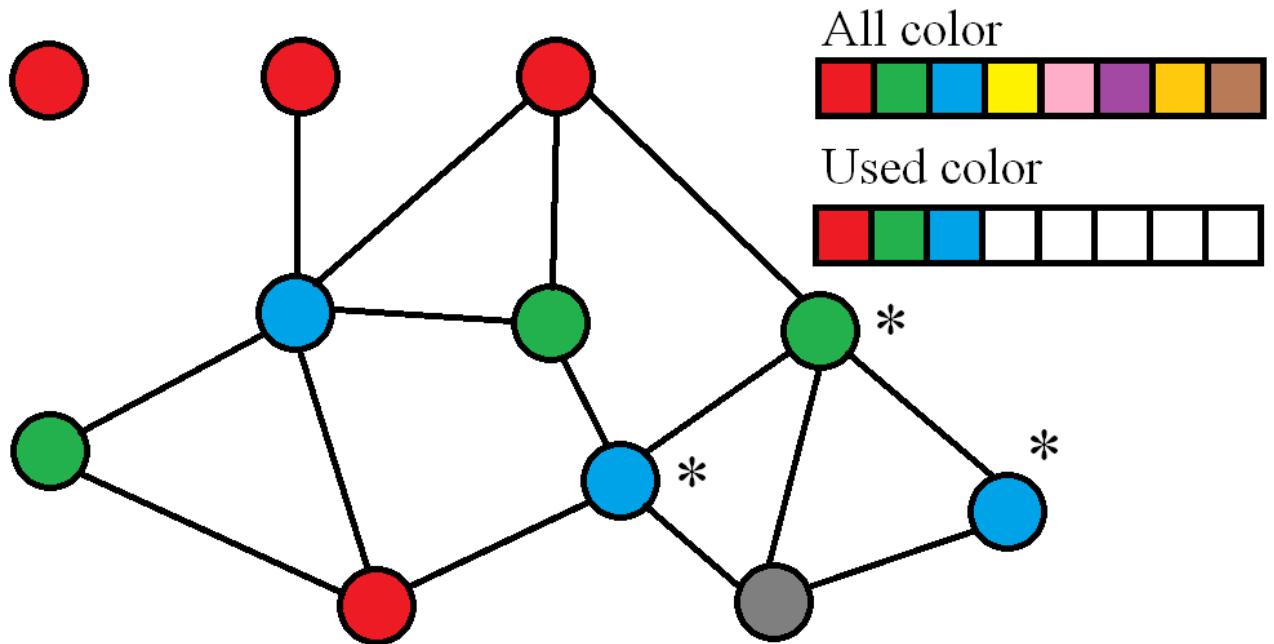
Далі фарбуємо обрану вершину в допустимий колір і вибираємо нову.



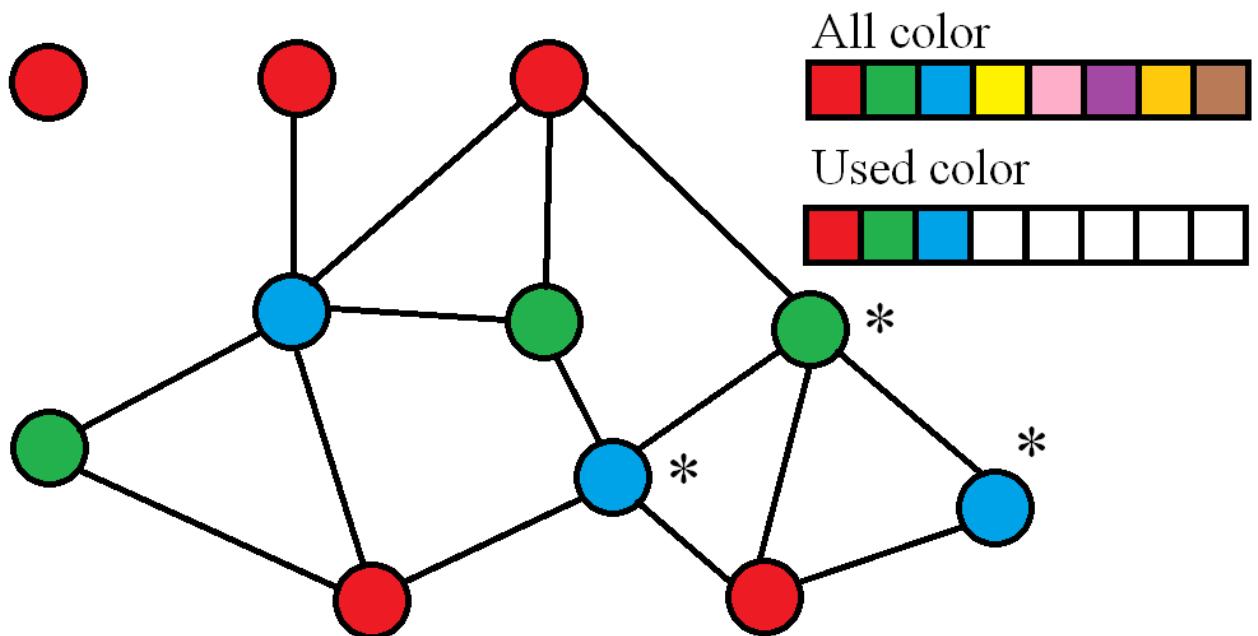
Далі повторюємо цю процедуру, важливо розуміти, що не можна вибрати тільки ті вершини, які вже були обрані, а не пофарбовані.



Алгоритм завершено, хроматичне число = 4, але якщо вибрati іншу вершину і інші кольори, то можна скоротити хроматичне число до 3.



Для цього і потрібен ітераційний процес.



5 ТЕМА 5 ТЕОРІЯ ІГОР

Теорія ігор - математичний метод вивчення оптимальних стратегій в іграх. Під грою розуміється процес, в якому беруть участь дві і більше сторін, які ведуть боротьбу за реалізацію своїх інтересів. Кожна зі сторін має свою мету і використовує деяку стратегію, яка може вести до виграшу або програшу - залежно від поведінки інших гравців. Теорія ігор допомагає вибрати найкращі стратегії з урахуванням уявлень про інших учасників, їх ресурсів і їх можливих вчинках.

Теорія ігор - розділ прикладної математики, точніше дослідження операцій. Найчастіше методи теорії ігор знаходять застосування в міжнародних відносинах, економіці, трохи рідше в інших суспільних науках - соціології, політології, психології, етиці, юриспруденції та інших. Починаючи з 1970-х років її взяли на озброєння біологи для дослідження поведінки тварин і теорії еволюції. Дуже важливе значення вона має для штучного інтелекту та кібернетики, особливо з проявом інтересу до інтелектуальних агентів.

Уявлення ігор

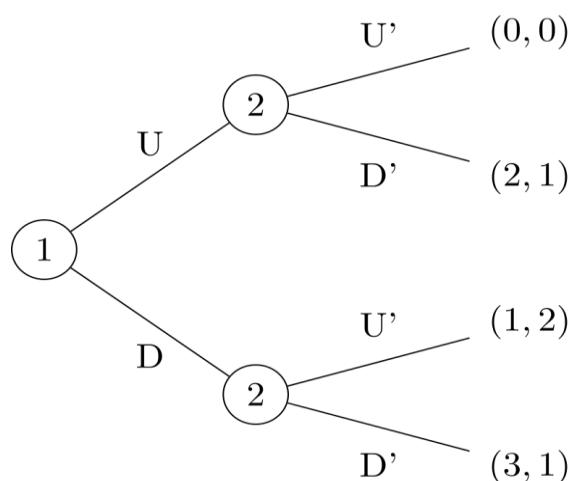
Ігри являють собою строго певні математичні об'єкти. Гра утворюється гравцями, набором стратегій для кожного гравця і вказівки виграшів, або **платежів**, гравців дляожної комбінації стратегій.

Більшість кооперативних ігор описуються характеристичною функцією, в той час як для інших видів частіше використовують нормальну або екстенсивну форму. Характеризуючі ознаки гри як математичної моделі ситуації:

- наявність декількох учасників;
- невизначеність поведінки учасників, пов'язана з наявністю у кожного з них декількох варіантів дій;
- відмінність (роздільність) інтересів учасників;
- взаємопов'язаність поведінки учасників, оскільки результат, одержуваний кожним з них, залежить від поведінки всіх учасників;

- наявність правил поведінки, відомих всім учасникам.

Розгорнутою формою гри називають її уявлення у вигляді дерева. Дерево складається з вершин і з'єднуючих їх ребер. Вершини підрозділяються на термінальні (кінцеві) і нетермінальні. Кожна нетермінальна вершина характеризується безліччю допустимих ходів і доступною для гравця інформацією. Термінальні вершини повідомляють про розмір виграшу, одержуваного по їх досягненні.



У нормальній, або стратегічній, формі: гра описується **платіжною матрицею**. Кожна сторона (точніше, вимір) матриці - це гравець, рядки визначають стратегії першого гравця, а стовпці - другого. На перетині двох стратегій можна побачити виграші, які отримають гравці. У прикладі праворуч, якщо гравець 1 обирає першу стратегію, а другий гравець - другу стратегію, то на перетині ми бачимо (-1, -1), це означає, що в результаті ходу обидва гравці втратили по одному очку.

	Гравець 2	Гравець 2
Стратегія 1		Стратегія 2

Гравець 1 Стратегія 1	4, 3	-1, -1
Гравець 1 Стратегія 2	0, 0	3, 4
<i>Нормальна форма для гри з 2 гравцями, у кожного з яких по 2 стратегії</i>		

Гравці обирали стратегії з максимальним для себе результатом, але програли через незнання ходу іншого гравця. Зазвичай в нормальній формі представляються ігри, в яких ходи робляться *одночасно*, або хоча б покладається, що всі гравці не знають про те, що роблять інші учасники. Такі ігри з **неповною інформацією** будуть розглянуті далі.

Типи ігор

Кооперативні та некооперативні

Гра називається кооперативною, або *коаліційною*, якщо гравці можуть об'єднуватися в групи, беручи на себе певні зобов'язання перед іншими гравцями і координуючи свої дії. Цим вона відрізняється від некооперативних ігор, в яких кожен зобов'язаний грati за себе. Розважальні ігри рідко є кооперативними, однак такі механізми нерідкі в повсякденному житті.

Часто припускають, що кооперативні ігри відрізняються саме можливістю спілкування гравців один з одним. У загальному випадку це невірно. Існують ігри, де комунікація дозволена, але гравці переслідують особисті цілі, і навпаки.

З двох типів ігор, некооперативні описують ситуації в найдрібніших деталях і видають більш точні результати. Кооперативні розглядають процес гри в цілому. Спроби об'єднати два підходи дали чималі результати. Так звана **програма Неша** вже знайшла вирішення деяких кооперативних ігор як ситуації рівноваги некооперативних ігор.

Гібридні ігри включають в себе елементи кооперативних і некооперативних ігор. Наприклад, гравці можуть утворювати групи, але гра буде проводитися в некооперативному стилі. Це означає, що кожен гравець буде переслідувати інтереси своєї групи, разом з тим намагаючись досягти особистої вигоди.

Симетричні і несиметричні

Гра буде симетричною тоді, коли відповідні стратегії у гравців будуть рівні, тобто матимуть однакові платежі. Інакше кажучи, якщо гравці можуть помінятись місцями і при цьому їх виграші за одній ті ж ходи не зміняться. Багато досліджуваних ігор для двох гравців - симетричні. Зокрема, такими є: «Дилема ув'язненого», «Полювання на оленя», «Яструби і голуби». В якості несиметричних ігор можна привести «Ультиматум» або «Диктатор».

У прикладі нижче гра на перший погляд може здатися симетричною через схожі стратегії, але це не так - адже виграш другого гравця за будь-якої зі стратегій (A, A) і (B, B) буде більше, ніж у першого.

	A	B
A	1, 2	0, 0
B	0, 0	1, 2
Несиметрична гра		

З нульовою сумою і з ненульовою сумою

Ігри з нульовою сумою - особливий різновид ігор з постійною сумою, тобто таких, де гравці не можуть збільшити або зменшити наявні ресурси, або фонд гри. В цьому випадку сума всіх виграшів дорівнює сумі всіх програшів при будь-якому ході. Подивіться на малюнок - числа означають платежі гравцям - і їх сума в кожній клітині дорівнює нулю. Прикладами таких ігор може служити покер, де один виграє всі ставки інших; реверсі, де захоплюються фішки супротивника; або банальна крадіжка.

Багато досліджуваних математиками ігор, в тому числі вже згадувана «Дилема ув'язненого», іншого роду: *в іграх з ненульовою сумою* виграш якогось

гравця не обов'язково означає програш іншого, і навпаки. Результат такої гри може бути менше або більше нуля. Такі ігри можуть бути перетворені до нульової суми - це робиться введенням **фіктивного гравця**, який «привласнює собі» надлишок або поповнює нестачу коштів.

Ще грою з відмінною від нуля сумою є торгівля, де кожен учасник отримує вигоду. Широко відомим прикладом, де вона зменшується, є війна.

	A	B
A	-1, 1	3, -3
B	0, 0	-2, 2
Гра з нульовою сумою		

Паралельні і послідовні

У паралельних іграх гравці ходять одночасно, або, принаймні, вони не обізнані про вибір інших до тих пір, поки всі не зроблять свій хід. У послідовних, або *динамічних*, іграх учасники можуть робити ходи в заздалегідь встановленому або випадковому порядку, але при цьому вони отримують деяку інформацію про попередні дії інших. Ця інформація може бути навіть не зовсім повною, наприклад, гравець може дізнатися, що його противник з десяти своїх стратегій точно не вибрав п'яту, нічого не дізnavши про інші.

Відмінності в поданні паралельних і послідовних ігор розглядалися вище. Перші зазвичай представляють в нормальній формі, а другі - в екстенсивній.

З повною або неповною інформацією

Важлива підмножина послідовних ігор складає ігри з повною інформацією. У такій грі учасники знають всі ходи, зроблені до поточного моменту, так само як і можливі стратегії супротивників, що дозволяє їм в деякій мірі передбачити подальший розвиток гри. Повна інформація недоступна в паралельних іграх, так як в них невідомі поточні ходи супротивників. Більшість досліджуваних в математиці ігор - з неповною

інформацією. Наприклад, вся суть Дилеми ув'язненого або Порівняння монеток полягає в їх неповноті.

У той же час є цікаві приклади ігор з повною інформацією: «Ультиматум», «Багатоніжка». Сюди ж відносяться шахи, шашки, го, манкала та інші.

Часто поняття повної інформації плутають зі схожим - **досконалої інформації**. Для останнього досить лише знання всіх доступних противникам стратегій, знання всіх їхніх ходів необов'язкове.

Прийняття оптимальних рішень в іграх

У цьому розділі йдеться про ігри з двома гравцями, які будуть називатися MAX і MIN з причин, які незабаром стануть очевидними. Гравець MAX ходить першим, після чого гравці роблять ходи по черзі до тих пір, поки гра не закінчиться. В кінці гри гравцеві-переможцю присвоюються очки, а на переможеного накладається штраф.

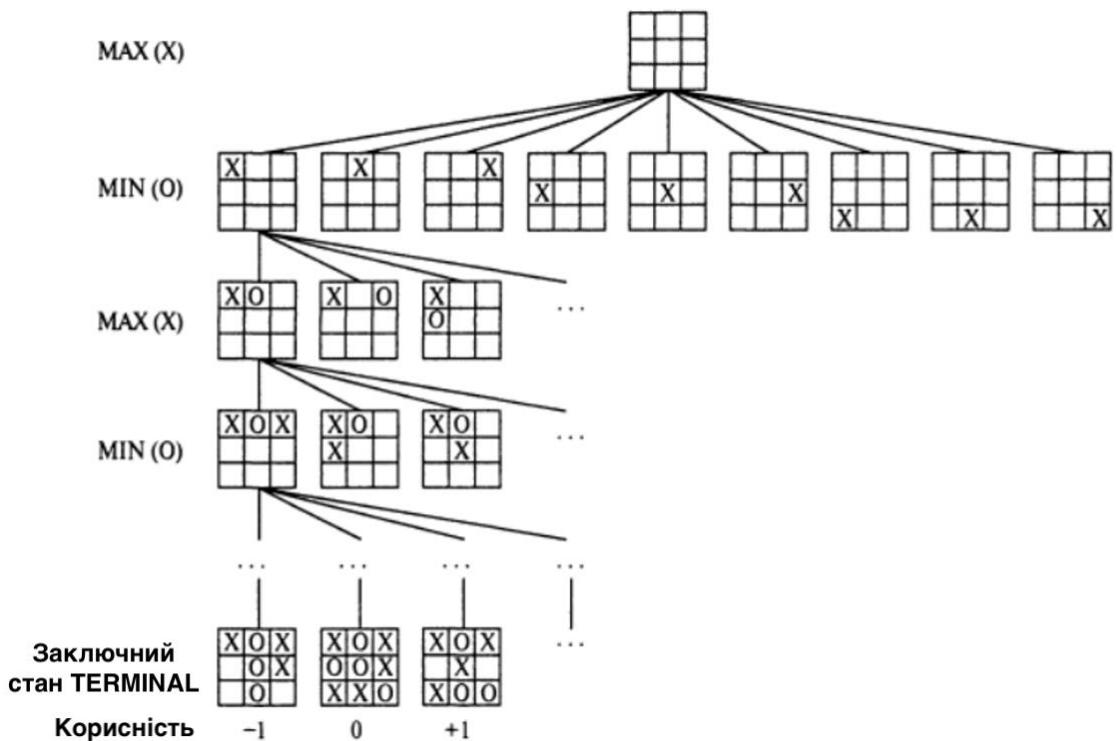
Гра може бути формально визначена як свого роду завдання пошуку з описаними нижче компонентами.

- **Початковий стан**, який включає позицію на дошці і визначає гравця, який повинен ходити.
- **Функція визначення наступника**, яка повертає список пар (move, state), кожна з яких вказує допустимий хід і результуючий стан.
- **Перевірка термінального стану**, яка визначає, що гра закінчена. Стани, в яких гра закінчена, називаються **термінальними станами**.
- **Функція корисності** (звана також *цільовою функцією*, або *функцією винагороди*), яка повідомляє числове значення термінальних станів. У шахах результатом є перемога, поразка чи нічия, зі значеннями +1, -1 або 0. Деякі ігри мають більш широкий діапазон можливих результатів; наприклад, кількість очок в нарди може становити від +192 до -192. В даному розділі в основному розглядаються ігри з нульовою сумою, хоча будуть також коротко згадуватися ігри з ненульовою сумою.

Початковий стан і допустимі ходи кожної сторони визначають **дерево гри** для даної гри. На [рис. 6. 1](#) показана частина дерева гри в хрестики-нулики. З початкового стану гравець MAX має дев'ять можливих ходів. Ходи чергуються так, що MAX ставить значок X, а MIN значок 0, до тих пір, поки не будуть досягнуті листові вузли, відповідні термінальним станам, таким, що один гравець поставив три своїх значки в один ряд або заповнені всі клітини. Число під кожним листовим вузлом вказує значення корисності відповідного термінального стану з точки зору гравця MAX; передбачається, що високі значення є сприятливими для гравця MAX і несприятливими для гравця MIN (саме тому в теорії цим гравцям присвоєні такі імена). Завдання гравця MAX полягає у використанні дерева пошуку (особливо даних про корисність термінальних станів) для визначення найкращого ходу.

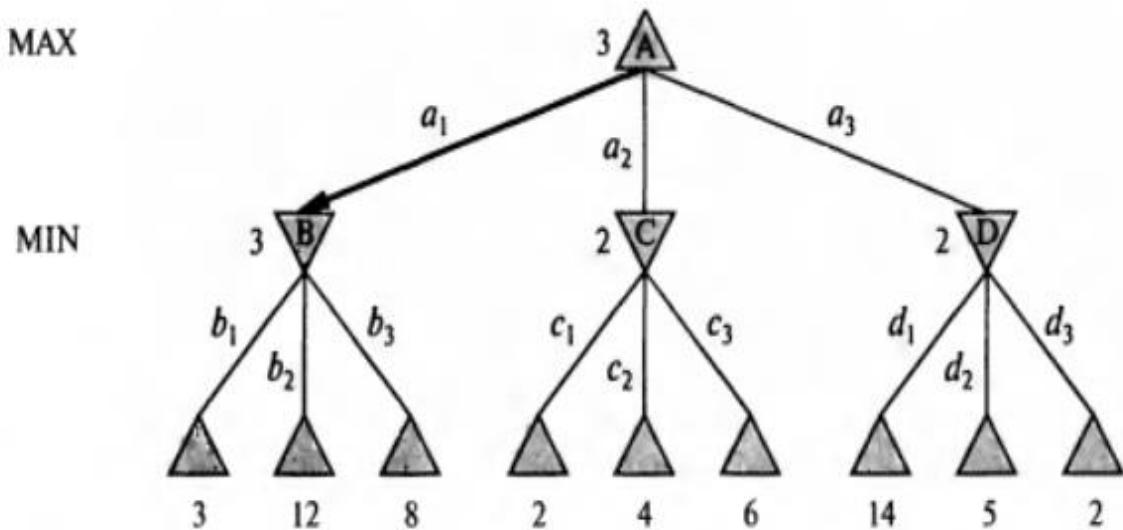
Оптимальні стратегії

При вирішенні звичайних завдань пошуку, оптимальне рішення для гравця MAX має являти собою послідовність ходів, що ведуть до мети - до термінального стану, який відповідає виграшу. З іншого боку, в грі бере участь також гравець MIN, який має іншу думку з цього приводу. Це означає, що гравець MAX повинен знайти надійну **стратегію**, що дозволяє визначити хід гравця MAX в початковому стані, потім ходи гравця MAX в станах, що стали результатом будь-якої можливої відповіді гравця MIN, а потім ходи MAX в станах, що стали результатом будь-якої можливої відповіді MIN на ті ходи, і т.д. Грубо кажучи, оптимальна стратегія призводить до підсумку, щонайменше, такому ж сприятливому, як і будь-яка інша стратегія, в тих умовах, коли доводиться грати з противником, що не допускає помилок. Перш за все розглянемо, як знайти цю оптимальну стратегію, навіть при тому, що для MAX часто буде нездійсненою задача її вичерпного обчислення в іграх, більш складних, ніж хрестики-нулики.



Мал. 6. 1. (Часткове) дерево пошуку для гри хрестики-нулики. Верхній вузол являє собою початковий стан, а першимходить гравець MAX , ставлячи значок X в порожній клітці. На цьому малюнку показана частина дерева пошуку, в якій демонструються ходи гравців MIN (значок O) і MAX (значок X), які чергуються. Ходи виконуються до тих пір, поки в кінцевому результаті не буде досягнуто одного з термінальних станів, яким можуть бути призначені дані про корисність відповідно до правил гри

Навіть такі прості ігри, як хрестики-нулики, є занадто складними, щоб можна було привести тут для них повне дерево гри, тому перейдемо до опису тривіальної гри, показаної на **рис. 6.2**. Можливі ходи гравця MAX з кореневого вузла позначені як a_1 , a_2 , і a_3 . Можливими відповідями на хід a_1 для гравця MIN є b_1 , b_2 , b_3 , і т.д. Дано конкретна гра закінчується після того, як кожен гравець, MAX і MIN , зроблять по одному ходу. (Відповідно до термінології теорії ігор, це дерево має глибину в один хід і складається зі зроблених двома учасниками ходів, кожен з яких називається **напівходом**.) Корисності термінальних станів в цій грі знаходяться в межах від 2 до 14.



Мал. 6.2 Дерево гри з двома напівходами. Вузли представляють собою "вузли MAX", в яких черга ходу належить гравцю MAX, а вузли розглядаються як "вузли MIN". Термінальні вузли показують значення корисності для MAX; інші вузли позначені їх мінімаксними значеннями. Кращим ходом гравця MAX від кореня є a_1 , оскільки веде до наступника з найбільшим мінімаксним значенням, а найкращою відповіддю MIN є b_1 , оскільки веде до наступника з найменшим мінімаксним значенням

При наявності дерева гри оптимальну стратегію можна визначити, досліджуючи **мінімаксне значення** кожного вузла, яке можна записати як Minimax-Value(n). Мінімаксним значенням вузла є корисність (для MAX) перебування у відповідному стані, за умови, що обидва гравці роблять ходи оптимальним чином від цього вузла і до вузла, що позначає кінець гри. Очевидно, що мінімаксним значенням термінального стану є просто його корисність. Більш того, якщо є вибір, гравець MAX повинен віддати перевагу ходу, що веде в стан з максимальним значенням, а гравець MIN – що веде в стан з мінімальним значенням. Тому має місце наведене нижче співвідношення

$$\text{Minimax-Value}(n) =$$

$$\begin{cases} \text{Utility}(n), \text{ якщо } n - \text{термінальний стан} \\ \max_{s \in \text{Successors}(n)} \text{Minimax-Value}(s), \text{ якщо } n - \text{вузол MAX} \\ \min_{s \in \text{Successors}(n)} \text{Minimax-Value}(s), \text{ якщо } n - \text{вузол MIN} \end{cases}$$

Застосуємо ці визначення до дерева гри, показаному на рис. 6.2. Термінальні вузли на нижчому рівні вже позначені числами, які вказують їх корисність. Перший вузол MIN, позначений як В, має трьох наступників зі значеннями 3, 12 і 8, тому його мінімаксне значення дорівнює 3. Аналогічним чином, інші два вузла MIN мають мінімаксне значення, рівне 2. Кореневим вузлом є вузол MAX; його наступники мають мінімаксні значення 3, 2 і 2, тому сам кореневий вузол має мінімаксне значення 3. Можна також визначити поняття **мінімаксного рішення**, прийнятого в корені дерева: дія а1 є оптимальним вибором для гравця MAX, оскільки веде до наступника з найвищим мінімаксним значенням.

У цьому визначенні оптимальної гри для гравця MAX передбачається, що гравець MIN також грає оптимальним чином: він максимізує результат, відповідний найгіршому результату гри для MAX. А що було б, якщо гравець MIN не грав оптимальним чином? В такому випадку можна легко показати, що гравець MAX домігся б ще більшого. Можуть існувати інші стратегії гри проти суперників, які грають неоптимальним чином, які дозволяють домогтися більшого, ніж мінімаксна стратегія; але ці стратегії обов'язково діють гірше проти суперників, які грають оптимально.

5.1 Мінімаксний алгоритм

Мінімаксний алгоритм (лістинг 6. 1) обчислює мінімаксне рішення з поточного стану. У ньому використовується просте рекурсивне обчислення мінімаксних значень кожного стану-наступника з безпосередньою реалізацією визначальних рівнянь. Рекурсія проходить всі рівні аж до листя дерева, а потім мінімаксні значення **резервуються** по всьому дереву в міру зворотного згортання рекурсії. Наприклад, в дереві, показаному на рис. 6.2, алгоритм спочатку виконав рекурсію з переходом на нижні рівні до трьох нижніх лівих вузлів і застосував до них функцію корисності Utility, щоб визначити, що значення корисності цих вузлів відповідно рівні 3, 12 і 8. Потім алгоритм визначає мінімальне з цих значень, 3, і повертає його в якості зарезервованого значення вузла В. Analogічний процес дозволяє отримати зарезервовані

значення 2 для вузла С і 2 для вузла D. Нарешті, береться максимальне зі значень 3, 2 і 2 для отримання зарезервованого значення 3 кореневого вузла.

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v  $\leftarrow -\infty$ 
    for a, s in SUCCESSORS(state) do v  $\leftarrow \text{MAX}(\text{v}, \text{MIN-VALUE}(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v  $\leftarrow \infty$ 
    for a, s in SUCCESSORS(state) do v  $\leftarrow \text{MIN}(\text{v}, \text{MAX-VALUE}(s))
    return v$$ 
```

Лістинг 6. 1. Алгоритм обчислення мінімаксних рішень. Він повертає дію, відповідну найкращому можливому ходу, тобто дію, що веде до результату з найкращою корисністю, відповідно до припущення, що противник грає з метою мінімізації корисності. Функції Max-Value і Min-Value використовуються для проходження через все дерево гри аж до листя, що дозволяє визначити **зарезервоване значення деякого стану**

Цей мінімаксний алгоритм виконує повне дослідження дерева гри в глибину. Якщо максимальна глибина дерева дорівнює m і в кожному стані є b допустимих ходів, то тимчасова складність цього мінімаксного алгоритму становить $O(b^m)$. Для алгоритму, який формує відразу всіх наступників, просторова складність дорівнює $O(bm)$, а для алгоритму, який формує наступників по одному, - $O(m)$. Безумовно, в реальних іграх такі витрати часу повністю неприйнятні, але даний алгоритм служить в якості основи математичного аналізу ігор, а також більш практичних алгоритмів.

5.2 Оптимальні рішення в іграх з кількома гравцями

Багато популярних ігор допускають наявність більше, ніж двох гравців. Розглянемо, як можна поширити ідею мінімаксного алгоритму на ігри з

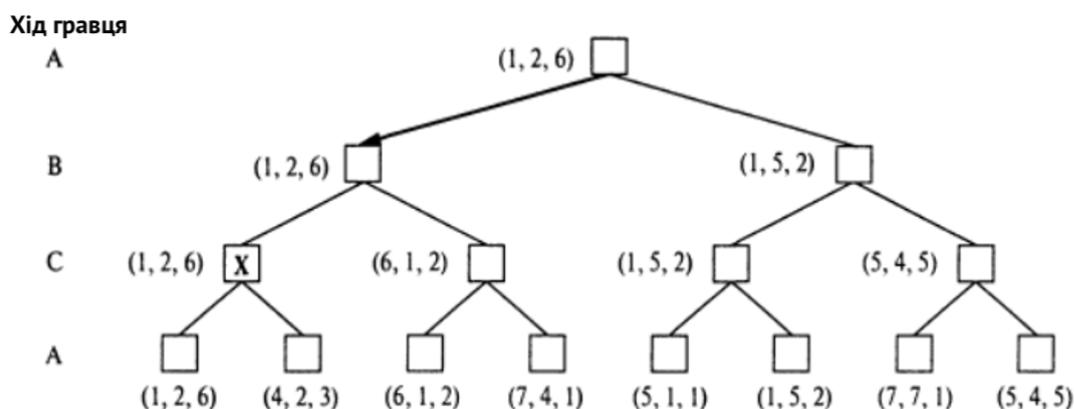
кількома гравцями. З точки зору технічної реалізації це зробити нескладно, але при цьому виникають деякі нові та цікаві концептуальні проблеми.

Спочатку необхідно замінити єдине значення для кожного вузла вектором значень. Наприклад, в грі з трьома гравцями, в якій беруть участь гравці A, B і C, з кожним вузлом асоціюється вектор $\langle V_a, V_b, V_c \rangle$. Для термінальних станів цей вектор задає корисність даного стану з точки зору кожного гравця. (В іграх з двома гравцями і нульовою сумою двоелементний вектор може бути скорочений до одного значення, оскільки значення в ньому завжди протилежні.) Найпростішим способом реалізації такого підходу є застосування функції Utility, яка повертає вектор значень корисності.

Тепер необхідно розглянути нетермінальні стани. Розглянемо вузол в дереві гри, показаному на рис. 6. 3, який позначений як X. В цьому стані гравець С обирає, що робити. Два варіанти ведуть до термінальних станів з векторами корисності $\langle V_a = 1, V_b = 2, V_c = 6 \rangle$ і $\langle V_a = 4, V_b = 2, V_c = 3 \rangle$. Оскільки 6 більше, ніж 3, гравець С повинен вибрати перший хід. Це означає, що якщо досягнуто стану X, то подальша гра приведе до термінального стану з корисностями $\langle V_a = 1, V_b = 2, V_c = 6 \rangle$. Отже, зарезервованим значенням X є цей вектор. Взагалі кажучи, зарезервоване значення вузла n являє собою вектор корисності такого вузла-наступника, який має найвище значення для гравця, що вибирає хід у вузлі n.

Будь-хто, хто грає в ігри з кількома гравцями, такі як DiplomacyTM («Дипломатія»), швидко дізнається, що в них відбувається набагато більше подій, ніж в іграх з двома гравцями. В іграх з кількома гравцями зазвичай створюються **альянси** між гравцями, або формальні, або неформальні. До того ж іноді в міру розвитку гри альянси то формуються, то руйнуються. Як можна зрозуміти таку поведінку? Чи є альянси природним наслідком вибору оптимальних стратегій для кожного гравця в грі з кількома гравцями? Як виявилося, вони дійсно можуть стати таким наслідком. Наприклад, припустимо, що гравці A і B мають слабкі позиції, а гравець C - більш сильну позицію. У такому випадку і для A, і для B часто буває оптимальним рішення

атакувати С, а не один одного, оскільки інакше С знищить кожного з них окремо. Таким чином, співпраця стає наслідком чисто егоїстичної поведінки. Безумовно, як тільки гравець С ослабне під спільним натиском, альянс втратить свій сенс і угоду може порушити або гравець А, або гравець В. В деяких випадках явно виражені альянси просто стають конкретним виразом того, що і так сталося б. А в інших випадках спроба порушити альянс викликає суспільний осуд, тому гравці повинні класти на терези негайно вигоду, яка досягається від порушення альянсу і довготривалий збиток, що виникає через те, що їх не будуть вважати такими, що заслуговують довіри.



Мал. 6.3. Перші три напівходи гри з трьома гравцями (A, B, C). Кожен вузол позначений значеннями, що досягаються з точки зору кожного участника. Найкращий хід позначений стрілкою, що виходить із кореня.

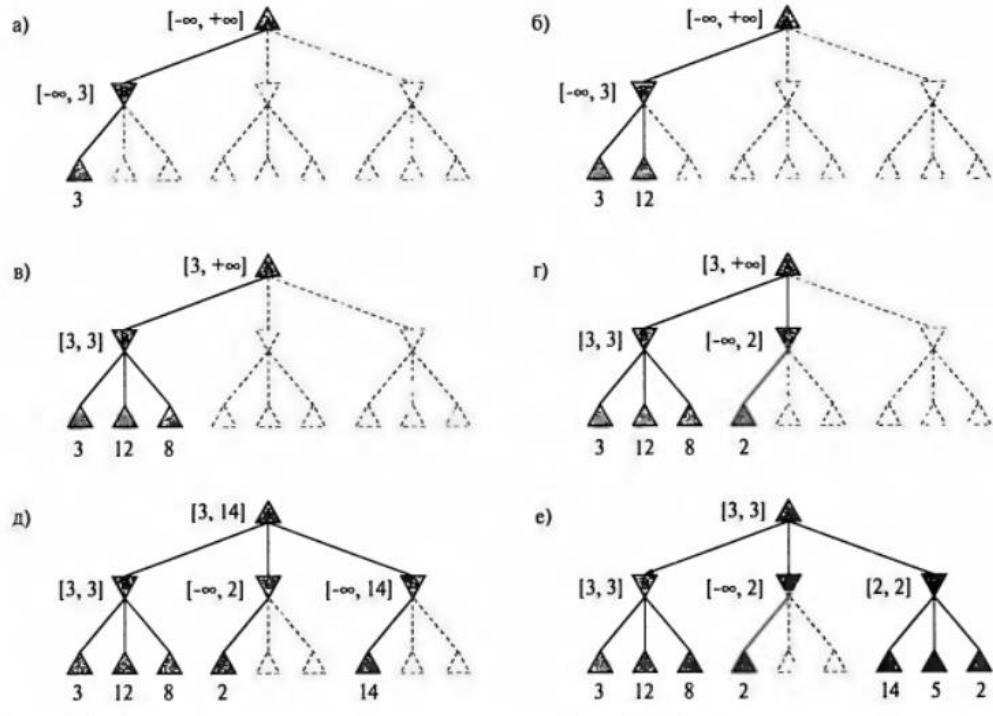
Якщо гра не має нульову суму, то співпраця може також виникати навіть при наявності всього двох гравців. Припустимо, що є термінальний стан з корисностями $\langle V_A = 1000, V_B = 1000 \rangle$ і що 1000 - максимальна можлива корисність для кожного гравця. В такому випадку оптимальна стратегія для обох гравців полягає в тому, щоб робити все можливе для досягнення цього стану; це означає, що гравці автоматично вступають в співпрацю для досягнення обопільно бажаної мети.

5.3 Алгоритм алфа-бета-відсікань

При мінімаксному пошуку проблема полягає в тому, що кількість станів гри, які повинні бути досліджені в процесі пошуку, залежить експоненціально

від кількості ходів. На жаль, таку експоненціальну залежність усунути неможливо, але фактично існує можливість скоротити її наполовину. Весь секрет полягає в тому, що обчислення правильного мінімаксного рішення можливе без перевірки кожного вузла в дереві гри. Це означає, що можна запозичити ідею **відсікання**, щоб виключити з розгляду великі частини дерева. Конкретний метод називається **альфа-бета-відсіканням**. Будучи застосований до стандартного мінімаксного дерева, цей метод повертає такі ж ходи, які повернув би мінімаксний метод, але відсікає гілки, по всій ймовірності, не здатні вплинути на остаточне рішення.

Знову розглянемо дерево гри з двома напівходами (див. рис. 6.2) і ще раз проведемо розрахунок оптимального рішення, на цей раз звертаючи особливу увагу на те, що відомо на кожному етапі цього процесу. Пояснення до даних етапів обчислення наведені на рис. 6.4. Результат полягає в тому, що мінімаксне рішення можна виявити, навіть не приступаючи до обчислення значень двох листових вузлів.



Мал. 6.4 Етапи обчислення оптимального рішення для дерева гри, показаного на рис. 6.2; в кожній точці відомий ряд можливих значень для кожного вузла: перший лист, розташований нижче вузла B , має значення 3. Тому B , який є вузлом MIN, має, найбільше, значення 3 (а); другий лист, розташований нижче вузла B , має значення 12 (б); гравець MIN

повинен уникати цього ходу, тому значення B , все ще найбільше, дорівнює 3; третій лист, розташований нижче вузла B , має значення 8; ми перевірили всіх наступників вузла B , тому значення B в точності дорівнює 3 (в). Тепер можна зробити висновок, що значення кореня, найменше, дорівнює 3, оскільки гравець MAX в корені робить вибір зі значенням 3; перший лист, що знаходиться нижче C , має значення 2. Тому C , який являє собою вузол MIN, має, найбільше, значення 2. Але відомо, що вузол B дозволяє досягти значення 3, тому гравець MAX ні в якому разі не повинен обирати вузол C . Це означає, що немає сенсу перевіряти інших наступників вузла C . Це - приклад застосування альфа-бета-відсікання (г). Перший лист, який перебуває нижче D , має значення 14, тому D має, найбільше, значення 14. Вони все ще вище, ніж найкраща альтернатива для гравця MAX (тобто 3), тому необхідно продовжити дослідження наступників вузла D . Слід також відзначити, що тепер визначені граничні значення всіх наступників кореневого вузла, тому значення кореня також дорівнює, найбільше, 14 (д); другий наступник D має значення 5, тому знову доводиться продовжувати дослідження. Значення третього наступника дорівнює 2, тому тепер значення D точно дорівнює 2. У кореневому вузлі гравець MAX приймає рішення зробити хід, що веде до вузла B , що дозволяє йому отримати значення 3(е)

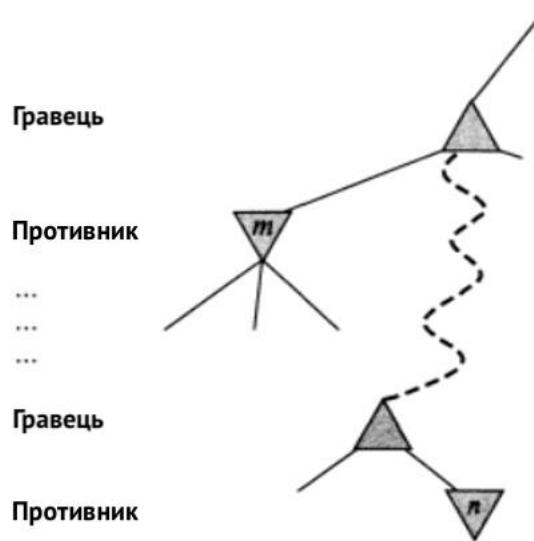
Цей підхід може також розглядатися під іншим кутом - як спрощення формул для отримання мінімаксного значення Minimax-Value. Припустимо, що два наступника вузла C на рис. 6.4, ще не оброблені в процесі обчислення, мають значення x і y , і припустимо, що z - мінімальне значення серед x і y . В такому випадку значення кореневого вузла можна знайти наступним чином:

$$\begin{aligned} \text{Minimax-Value (root)} &= \max (\min (3, 12, 8), \min (2, x, y), \min (14, 5, 2)) \\ &= \max (3, \min (2, x, y), 2) \\ &= \max (3, z, 2) \text{ де } z \leq 2 \\ &= 3 \end{aligned}$$

Іншими словами, значення кореневого вузла, а отже, і мінімаксне рішення не залежить від значень відсічених листових вузлів x і y .

Альфа-бета-відсікання може застосовуватися до дерев будь-якої глибини; до того ж часто виникає можливість відсікати цілі піддерева, а не просто листя. Загальний принцип полягає в наступному: розглянемо вузол n , що знаходиться де-небудь в дереві (рис. 6.5), такий, що учасник гри з боку спостерігача (назвемо його Гравець) має можливість вибрати хід, що веде до

цього вузла. Але якщо Гравець має кращий вибір t або в батьківському вузлі вузла n , або в будь-якій точці вибору, що знаходиться ще вище в дереві, то вузол n ніколи не буде досягнутий в грі, яка відбувається в дійсності. Тому після отримання достатньої інформації про вузол n (шляхом дослідження деяких з його нащадків) для того, щоб з повною впевненістю прийти до цього висновку, можна виконати його відсікання.



Мал. 6.5. Альфа-бета-відсікання: загальний випадок. Якщо для Гравця вузол t краще, ніж n , то вузол n ніколи не зустрінеться в грі

Нагадаємо, що мінімаксний пошук здійснюється в глибину, тому в будь-який момент часу достатньо розглядати вузли вздовж єдиного шляху в дереві. Алгоритм альфа-бета-відсікання отримав свою назву за такими двома параметрами, які представляють межі в зарезервованих значеннях, присутніх у всіх вузлах уздовж цього шляху:

- α = значення найкращого варіанту (тобто варіанту з найвищим значенням), який був досі знайдений в будь-якій точці вибору вздовж шляху для гравця MAX;
- β = значення найкращого варіанту (тобто варіанту з найнижчим значенням), який був досі знайдений в будь-якій точці вибору вздовж шляху для гравця MIN.

Алгоритм альфа-бета-пошуку в процесі своєї роботи оновлює значення α і β , а також відсікає гілки, які залишилися у вузлі (тобто припиняє рекурсивні виклики), як тільки стає відомо, що значення поточного вузла гірше в порівнянні з поточним значенням α або β для гравця MAX або MIN відповідно. Повний алгоритм приведений в лістингу 6.2. Рекомендуємо читачеві простежити за його поведінкою стосовно дерева, показаного на рис. 6.4.

Лістинг 6.2. Алгоритм альфа-бета-пошуку. Зверніть увагу на те, що процедури, які тут застосовуються, залишаються такими ж, як і процедури алгоритму Minimax, наведеного в лістингу 6.1, за винятком двох рядків, введених як в процедуру Min-Value, так і в Max-Value, які супроводжують значення α і β (а також виконують відповідні дії щодо подальшої передачі цих параметрів)

```

function ALPHA-BETA-DECISION(state) returns an action
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))



---


function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
        α, the value of the best alternative for MAX along the path to state
        β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v  $\leftarrow -\infty
    for a, s in SUCCESSORS(state) do
        v  $\leftarrow \text{MAX}(\text{MIN-VALUE}(s, α, β))
        if v  $\geq \beta$  then return v
        α  $\leftarrow \text{MAX}(\alpha, v)
    return v



---


function MIN-VALUE(state, α, β) returns a utility value
    same as MAX-VALUE but with roles of  $\alpha, \beta$  reversed$$$ 
```

Ефективність алгоритму альфа-бета-відсікання у вищій мірі залежить від того, в якому порядку відбувається перевірка наступників. Наприклад, на рис. 6.4, де неможливо було б взагалі виконати відсікання будь-яких наступників вузла D, оскільки в першу чергу були б сформовані найгірші наступники (з точки зору гравця MIN). А якби в першу чергу був сформований третій наступник, то була б можливість відсікти двох інших. На підставі цього можна

зробити висновок, що має сенс прагнути досліджувати в першу чергу таких наступників, які, цілком ймовірно, можуть стати найкращими.

Якщо прийняти припущення, що це може бути зроблено, то виявиться, що в алгоритмі альфа-бета-відсікання для визначення найкращого ходу достатньо досліджувати тільки $O(B^{m/2})$ вузлів, а не $O(B^m)$ вузлів, як при використанні мінімаксного алгоритму. Це означає, що ефективний коефіцієнт розгалуження стає рівним), (\sqrt{b}) а не b ; наприклад, для шахів він дорівнює 6, а не 35. Іншими словами, за такий же час альфа-бета-пошук дозволяє заглянути в дерево гри приблизно в два рази далі в порівнянні з мінімаксним пошуком. А якщо дослідження наступників відбувається у випадковому порядку, а не за принципом першочергового вибору найкращих варіантів, то при помірних значеннях b загальна кількість досліджених вузлів буде становити приблизно $O(B^{3m/4})$. У разі шахів застосування досить простої функції впорядкування (наприклад, такої, в якій в першу чергу розглядаються взяття фігур, потім загрози, потім ходи вперед, а після цього ходи назад) дозволяє залишатися в межах, що не перевищують подвійне значення результату $O(B^{m/2})$, який може бути отриманий в найкращому разі. Додавання динамічних схем упорядкування ходів, зокрема, таких, в яких в першу чергу перевіряються ходи, позначені як найкращі на попередньому етапі, дозволяють підійти зовсім близько до цієї теоретичної межі.

5.4 Неідеальні рішення, що приймаються в реальному часі

Мінімаксний алгоритм формує весь простір пошуку гри, а алгоритм альфа-бета-відсікання дозволяє відсікати значні його частини. Проте при використанні алгоритму альфа-бета-відсікання все ще доводиться виконувати пошук аж до термінальних станів, принаймні, в деякій області простору пошуку. Зазвичай завдання досягнення такої глибини на практиці нездійсненне, оскільки ходи повинні бути зроблені за якийсь прийнятний час, як правило, не більше, ніж за кілька хвилин. Замість цього в статті Шенона *Programming a computer for playing chess* від 1950 року було запропоновано, щоб програми припиняли пошук завчасно і застосовували до станів якусь

евристичну **функцію оцінки**, по суті, перетворюючи нетермінальні вузли в термінальні листя. Іншими словами, в цій статті було запропоновано модифікувати мінімаксний пошук або альфа-бета-пошук у двох відносинах: замінити функцію корисності евристичною функцією оцінки *Eval*, яка дає оцінку корисності даної позиції, а перевірку термінальної позиції замінити **перевіркою зупинки**, яка дозволяє вирішити, коли слід застосовувати функцію *Eval*.

Функція оцінки

Функція оцінки повертає прогноз очікуваної корисності гри з даної конкретної позиції за аналогією з тим, як евристичні функції, описані в розділі 4, повертають прогнозоване значення відстані до цілі. Ідея такого "оцінювача" в той час, коли Шенон запропонував нею скористатися, була не нова. Протягом багатьох століть шахісти (і шанувальники інших ігор) розробили способи вироблення суджень про вартість позиції, оскільки люди ще більш обмежені в обсягах пошуку, який може бути ними виконаний, ніж комп'ютерні програми. **Повинно бути очевидно, що продуктивність будь-якої програми ведення гри залежить від якості функції оцінки, яка застосовується.** Неточна функція оцінки призведе агента до позицій, які виявляться програшними. Тому виникає важливe питання - як саме слід проектувати хороші функції оцінки?

По-перше, функція оцінки повинна впорядковувати термінальні стани таким же чином, як і справжня функція корисності; в іншому випадку агент, який її використовує, може обрати неоптимальні ходи, навіть володіючи здатністю прораховувати всі ходи до кінця гри. По-друге, обчислення не повинні займати занадто багато часу! (У функції оцінки можна було б викликати Minimax-Decision в якості процедури і обчислювати точну вартість даної позиції, але це поставило б під сумнів те, до чого ми прагнемо, - економію часу.) По-третє, для нетермінальних станів значення цієї функції оцінки повинні строго корелювати з фактичними шансами на виграв.

Вираз «шанси на виграв» на перший погляд може здатися дивним. Зрештою, шахи - це ж не гра з елементами випадковості: в ній безумовно

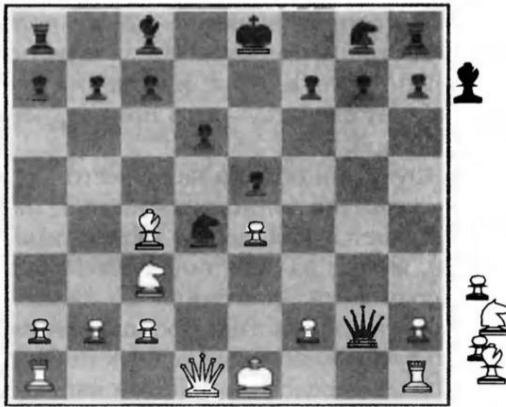
відомо поточний стан і для визначення наступного ходу не потрібно кидати жереб. Але якщо пошук повинен припинятися в нетермінальних станах, то в даному алгоритмі буде обов'язково залишатися невизначеність щодо остаточних результатів для цих станів. Невизначеність такого роду викликана обчислювальними, а не інформаційними обмеженнями. Через обмежений обсяг обчислень, які дозволено виконати в функції оцінки для даного конкретного стану, краще, що вона може зробити, - це прийняті якесь припущення щодо кінцевого результату.

На практиці для проведення аналізу такого роду потрібно враховувати дуже багато категорій і тому накопичити занадто багато досвіду, щоб можна було оцінити всі ймовірності виграшу. Замість цього в більшості функцій оцінки обчислюються окремі представлені в числовому вигляді значення вкладу, які залежать від кожної характеристики, після чого ці значення комбінуються для пошуку сумарного значення. Наприклад, в підручниках з шахів для початківців можна знайти наближені оцінки **вартості матеріалу** дляожної фігури: наприклад, такі, що пішак має вартість 1, кінь або слон - 3, тура - 5, а ферзь - 9. Інші характеристики, такі як «хороша пішакова структура» і «безпека короля», можуть оцінюватися як рівні, скажімо, половині вартості пішака. Після цього вартості таких характеристик просто складаються для отримання оцінки позиції. Надійна перевага, еквівалентна вартості пішака, розцінюється як значна ймовірність виграшу, а надійна перевага в три пішаки має майже напевно забезпечити перемогу, як показано [на рис. 6.6, а.](#)

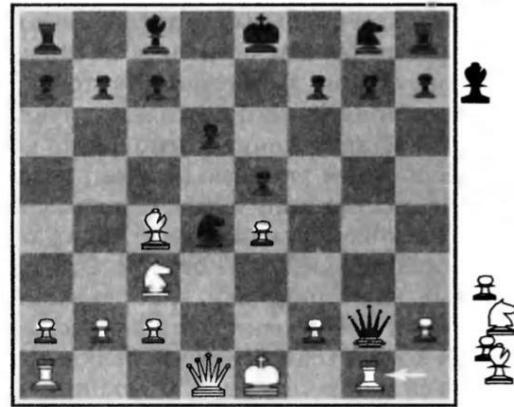
В математиці функція оцінки такого типу називається **зваженою лінійною функцією**, оскільки вона може бути представлена наступним чином:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s) = \sum_{i=1}^n w_i f_i(s)$$

де кожен коефіцієнт являє собою вагу, а кожна функція оцінює деяку характеристику позиції. У шахах функція може визначати кількість на дошці фігур кожного виду, а коефіцієнт - оцінювати вартості цих фігур (1 за пішака, 3 за слона і т.д.).



a) Хід білих



б) Хід білих

Мал. 6.6 Дві і трохи різні шахові позиції: чорні мають перевагу в одного коня і двох пішаків і повинні виграти (а); чорні програють після того, як білі беруть ферзя (б)

На перший погляд метод обчислення суми вартостей характеристик може здаватися прийнятним, але в дійсності він заснований на дуже радикальному допущенні, що внесокожної характеристики не залежить від вартості інших характеристик. Наприклад, привласнюючи слону вартість 3, ми ігноруємо той факт, що слони стають більш потужними в кінці гри, коли мають великий обсяг простору для маневру. З цієї причини в сучасних програмах для шахів і інших ігор використовуються також нелінійні комбінації характеристик. Наприклад, пара слонів може коштувати трохи більше в порівнянні з подвоєною вартістю одного слона, а слон коштує трохи більше в кінці гри, ніж на початку.

Уважний читач повинен був помітити, що всі ці характеристики і ваги не входять до складу шахових правил! Вони були вироблені протягом століть на основі досвіду гри людей в шахи. Застосування цих характеристик і ваг на основі описаної лінійної форми оцінки дозволяє домогтися найкращої апроксимації по відношенню до істинного упорядкування станів за вартістю. Зокрема, досвід показує, що надійна матеріальна перевага більше, ніж в один пункт, цілком ймовірно, призводить до виграшу при всіх інших рівних умовах; перевага в три пункти є достатньою майже для безумової перемоги. У таких іграх, де досвід зазначеного виду відсутній, ваги функції оцінки можуть бути отримані за допомогою методів машинного навчання, наведених у розділі 18. Є обнадійливим той факт, що застосування зазначених методів до шахів підтвердило, що слон дійсно має вартість, приблизно рівну трьом пішакам .

Припинення пошуку

Наступний етап полягає в тому, що алгоритм *Alpha-Beta-Search* повинен бути модифікований так, щоб він викликав евристичну функцію *Eval*, коли виникає необхідність зупинити пошук. З точки зору реалізації необхідно замінити два рядки в лістингу 6.2, в яких згадується функція *Terminal-Test*, наступним рядком:

```
if Cutoff-Test (state, depth) then return Eval (state)
```

Необхідно також передбачити виконання певних технічних операцій для того, щоб поточне значення глибини *depth* нарощувалось при кожному рекурсивному виклику. Найбільш прямолінійний підхід до управління об'ємом пошуку полягає в тому, що повинна встановлюватися фіксована межа глибини, щоб функція *Cutoff-Test (state, depth)* повертала значення *true* при всіх значеннях *depth*, що перевищують деяку фіксовану глибину *d*. (Вона повинна також повертати *true* для всіх термінальних станів, як було передбачено і в функції *Terminal-Test*.) Глибина *d* обрана таким чином, щоб час, що використовується не перевищував допустимий за правилами гри.

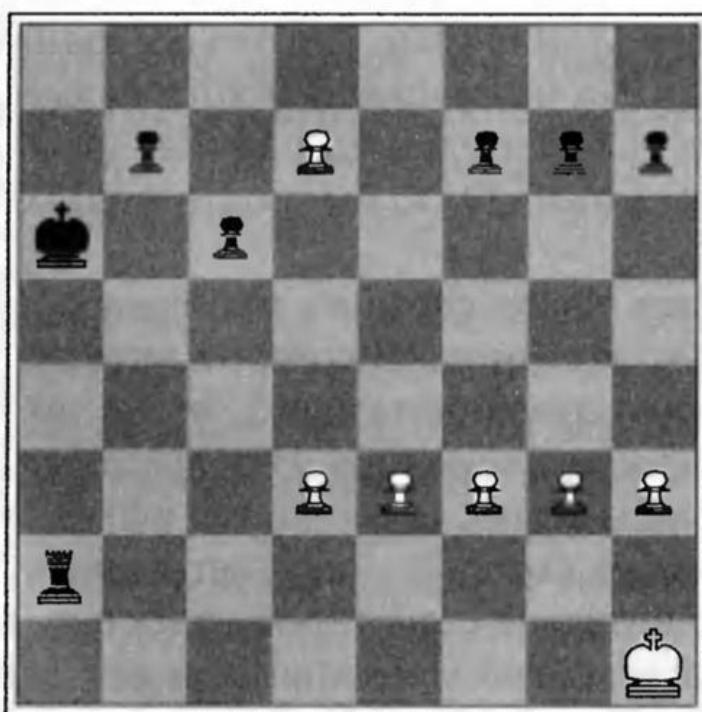
Більш надійний підхід полягає у використанні методу ітеративного поглиблення, який визначений в розділі 3. Після закінчення відведеного часу програма повертає хід, обраний за підсумками найбільш глибокого завершеного пошуку. Однак подібні підходи можуть призводити до помилок, обумовлених наближеним характером функції оцінки. Ще раз розглянемо просту функцію оцінки для шахів, засновану на обліку переваги в матеріалі. Припустимо, що програма виконує пошук до межі глибини, досягаючи позиції, наведеної на [рис. 6.6, б](#), де чорні мають перевагу на одного коня і дві пішаки. Програма повідомила б про це як про евристичне значення цього стану, оголосивши тим самим, що цей стан, цілком ймовірно, приведе до перемоги чорних. Але на наступному ходу білі беруть ферзя чорних без компенсації. Тому в дійсності дана позиція є виграшною для білих, але про це можна було б дізнатися, тільки заглянувши вперед ще на один напівхід.

Очевидно, що потрібна більш складна перевірка зупинки. Функція оцінки повинна застосовуватися тільки до позицій, які є **спокійними**, тобто характеризуються низькою ймовірністю того, що в них у найближчому майбутньому відбудуться різкі зміни у вартості. Наприклад, в шахах такі позиції, в яких можуть бути зроблені бажані взяття фігур, не є спокійними для такої функції оцінки, в якій враховується лише матеріал. Неспокійні позиції можуть бути додатково розгорнуті до тих пір, поки не будуть досягнуті спокійні позиції. Подібний додатковий пошук називається **пошуком спокійних позицій**; іноді він обмежується тим, що в ньому розглядаються тільки ходи певних типів, такі як ходи зі взяттям фігур, що дозволяє швидко усувати всі невизначеності в цій позиції.

Завдання усунення **ефекту горизонту** є більш складним. Цей ефект виникає, якщо програма стикається з якимось ходом противника, який заподіює серйозної шкоди і в кінцевому підсумку є неминучим. Розглянемо шахову позицію, наведену на рис. 6.7. Чорні перевершують білих за кількістю матеріалу, але якщо білі зможуть просунути свою пішака з сьомої горизонтали на восьму, то пішак стане ферзем і забезпечить легку перемогу білих. Чорні можуть відстрочити цей підсумок на 14 напівходів, оголошуючи шах білим за допомогою тури, але пішак неминуче стане ферзем. Одним з недоліків пошуку на фіксовану глибину є те, що алгоритм, який застосовується при цьому, не дозволяє визначити, що такі відволікаючі ходи не здатні запобігти ходу з перетворенням пішака на ферзя. У цьому випадку прийнято вважати, що відволікаючі ходи виводять неминучий хід з перетворенням пішака на ферзя "за межі горизонту пошуку" - в те місце, де небезпечний хід неможливо виявити.

У міру того як вдосконалення апаратних засобів, що застосовуються для ведення гри в шахи, призводить до збільшення глибини пошуку, стає все більш можливим те, що ефект горизонту буде виникати не так часто, оскільки дуже довгі послідовності ходів, що дозволяють відстрочити виконання небажаного ходу, виникають вкрай рідко. Для запобігання ефекту горизонту без занадто значного збільшення вартості пошуку виявилося також вельми ефективним використання **одинарних розширень**. Одинарним розширенням називається

хід, який "безумовно краще" в порівнянні з усіма іншими ходами в даній конкретній позиції. Пошук з одинарним розширенням дозволяє вийти за звичайні межі глибини пошуку без внесення значних витрат, оскільки в ньому коефіцієнт розгалуження дорівнює 1. (Пошук спокійної позиції може розглядатися як один з варіантів одинарних розширень.) На рис. 6.7. пошук з одинарним розширенням дозволяє знайти виконуваний в кінцевому підсумку хід перетворення пішака на ферзя, за умови, що ходи чорних з оголошенням шаха і ходи білих королем можуть бути визначені як "безумовно кращі" у порівнянні з іншими варіантами.



Хід чорних

Мал. 6.7 Прояв ефекту горизонту. Серія ходів чорною турою призводить до примусового висновку неминучого ходу білих з перетворенням пішака на ферзя «за горизонт», в зв'язку з чим ця позиція починає здаватися такою, яка допускає виграш чорних, тоді як фактично вона є виграшною для білих

До цих пір мова йшла про те, що припинення пошуку на певному рівні і виконання альфа-бета-відсікання, мабуть, не впливає на результат. Існує також можливість виконувати **попереднє відсікання**, а це означає, що деякі ходи в даному конкретному вузлі відсікаються негайно, без подальшого розгляду.

Очевидно, що більшість людей, які грають в шахи, розглядають лише кілька ходів з кожної позиції (принаймні, свідомо). На жаль, цей підхід є досить небезпечним, оскільки немає ніякої гарантії того, що не відбудеться відсікання кращого ходу. А якщо відсікання застосовується недалеко від кореня, результат може виявитися катастрофічним, оскільки занадто часто виникають такі ситуації, що програма пропускає деякі "очевидні" ходи. Попереднє відсікання може використовуватися безпечно в особливих ситуаціях (наприклад, якщо два ходи є симетричними або еквівалентними з якихось інших ознак, то необхідно розглядати тільки один з них) або при аналізі вузлів, які знаходяться глибоко в дереві пошуку.

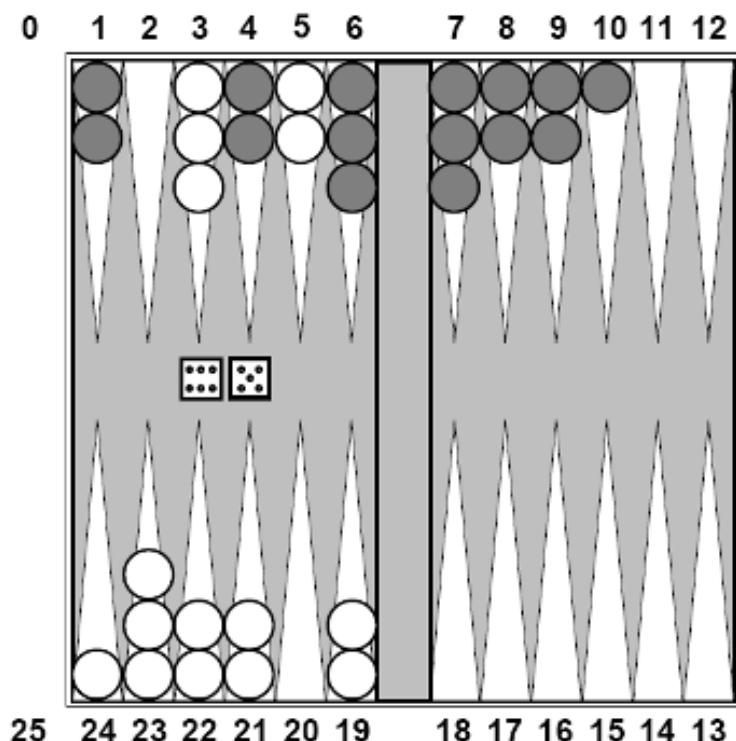
В результаті спільногого використання всіх методів, описаних вище, з'являється можливість створити програму, яка непогано грає в шахи (або інші ігри). Припустимо, що реалізована функція оцінки для шахів, передбачена розумна перевірка зупинки в поєднанні з пошуком спокійної позиції, а також передбачена велика таблиця транспозицій. Крім того, припустимо, що, витративши довгі місяці на скрупульозну розробку програм, що функціонують на рівні бітів, ми отримали можливість формувати й оцінювати приблизно мільйон вузлів в секунду на новітньому персональному комп'ютері, що дозволяє виконувати пошук приблизно серед 200 мільйонів вузлів в розрахунку на кожен хід при стандартному контролі часу (три хвилини на кожен хід). Коефіцієнт розгалуження для шахів становить в середньому приблизно 35, а 35^5 дорівнює приблизно 50 мільйонам, тому при використанні мінімаксного пошуку ми отримаємо можливість заглядати вперед лише приблизно на п'ять напівходів. Така програма, хоча і не зовсім некомпетентна, може бути легко обдурена людиною, гравцем в шахи середнього рівня, який іноді здатний планувати на шість або вісім напівходів вперед. Альфа-бета-пошук дозволяє досягти глибини приблизно в 10 напівходів, що призводить до посилення гри до рівня майстра. У розділі 6.7 описані додаткові методи відсікання, які дозволяють збільшити ефективну глибину пошуку приблизно до 14 напівходів. Щоб досягти рівня гросмейстера, потрібно ретельно налаштована функція оцінки і велика база даних із записами оптимальних ходів в дебюті та ендшпілі.

Не завадило б також мати суперкомп'ютер для експлуатації на ньому такої програми!

5.5 Ігри, які включають елемент випадковості

У реальному житті відбувається багато непередбачуваних зовнішніх подій, через які люди потрапляють в непередбачені ситуації. Ця непередбачуваність відбувається в багатьох іграх за рахунок включення елемента випадковості, такого як метання жереба. Таким чином, ігри з елементом випадковості дозволяють нам на один крок наблизитися до реальності і тому має сенс розглянути, як це відбувається на процесі прийняття рішень.

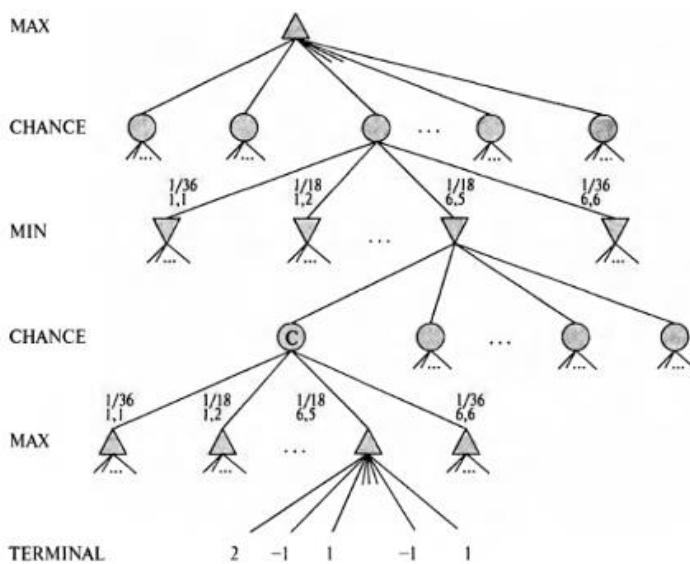
Однією з типових ігор, в яких поєднується удача та мистецтво, є нарди. Перед кожним своїм ходом гравець кидає кубики для визначення допустимих ходів. Наприклад, в позиції гри в нарди, яка наведена на рис. 6.8, білим випали очки 6-5 і вони мають чотири можливих ходи.



Мал. 6.8. Типова позиція гри в нарди. Мета гри полягає в тому, щоб зняти з дошки всі фішки. Білі ходять за годинниковою стрілкою до поля 25, а чорні - проти годинникової стрілки до поля 0. Фішка може переходити на будь-яке поле, якщо на ньому не знаходитьсь

декілька фішок противника; якщо ж на цьому полі є лише одна фішка противника, вона потрапляє в полон і повинна почати свій рух з самого початку. У показаній тут позиції білі після метання жереба отримали очки 6-5 і повинні вибирати серед чотирьох допустимих ходів: (5-10, 5-11), (5-11, 19-24), (5-10, 10-16) і (5-11, 11-16)

Хоча білим відомо, які їхні допустимі ходи, вони не знають, які очки принесе метання жереба чорним, і тому не можуть визначити, якими будуть допустимі ходи чорних. Це означає, що білі не мають можливості сформувати стандартне дерево гри такого типу, яке зустрілося нам в шахах, а також хрестиках-нуликах. Древо гри в нарди, крім вузлів MAX і MIN, має включати **вузли жеребкування**. Вузли жеребкування позначені на рис. 6.9 колами. Гілки, що ведуть від кожного вузла жеребкування, позначають можливі результати метання жереба, тому кожна з них відзначена написом із зазначенням кількості очок і ймовірності, з якою можуть бути отримані ці очки. Існує 36 різних сполучень очок на двох кубиках, і всі вони є рівноЯмовірними; але оскільки такі поєднання очок, як 6-5 і 5-6, є однаковими, є тільки 21 помітне поєднання очок. Ймовірність появи шести дублів (від 1-1 до 6-6) дорівнює $1/36$, а кожне з 15 інших різних сполучень очок має ймовірність $1/18$.



очікуване значення, в якому очікуваний результат встановлюється з урахуванням всіх можливих випадінь жереба, які можуть статися. Це призводить до узагальнення **мінімаксного значення** для детермінованих ігор до **очікуваного мінімаксного значення** (expectimax value) для ігор з вузлами жеребкування. Термінальні вузли і вузли MAX і MIN (для яких відомі результати жеребкування) застосовуються так само, як і раніше, а вузли жеребкування оцінюються шляхом отримання зваженого середнього значень, отриманих в результаті всіх можливих випадінь жереба, тобто наступним чином:

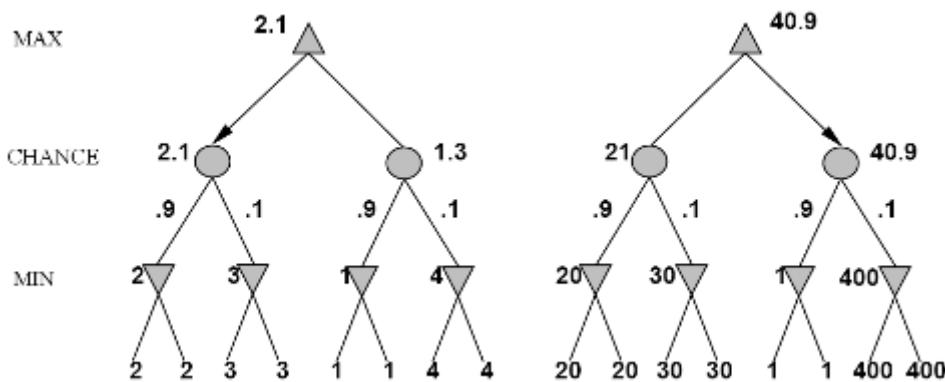
$$\text{Expectminimax}(n) = \begin{cases} \text{Utility}(n), \text{ якщо } n - \text{термінальний стан} \\ \max_{s \in \text{Successors}(n)} \text{Expectminimax}(s), \text{ якщо } n - \text{вузол MAX} \\ \min_{s \in \text{Successors}(n)} \text{Expectminimax}(s), \text{ якщо } n - \text{вузол MIN} \\ \sum_{s \in \text{Successors}(n)} P(s) \cdot \text{Expectminimax}(s), \text{ якщо } n - \text{вузол жеребкування} \end{cases}$$

де функція визначення наступника для вузла жеребкування n просто доповнює стан n кожним можливим випаданням жереба для формування кожного наступника s , а $P(s)$ - ймовірність, з якою відбувається випадання жереба. Результати обчислення цих рівнянь можуть резервуватися рекурсивно у всіх вузлах аж до кореня дерева так само, як і в мінімаксному алгоритмі. Ми залишаємо читачеві опрацювання всіх деталей цього алгоритму в якості вправи.

Оцінка позиції в іграх з вузлами жеребкування

За аналогією з мінімаксними значеннями, очевидний підхід до використання очікуваних мінімаксних значень полягає в тому, щоб зупиняти пошук в деякій точці і застосовувати функцію оцінки до кожного листу. На перший погляд може здатися, що функції оцінки для таких ігор, як нарди, повинні бути повністю подібними функціям оцінки для шахів, адже від них вимагається лише те, щоб вони привласнювали вищі оцінки найкращим

позиціям. Але в дійсності наявність вузлів жеребкування означає, що потрібен більш ретельний аналіз змісту таких оціночних значень. На рис. 6.10 показано, що відбувається в іграх з елементами випадковості - при використанні функції оцінки, яка присвоює листам значення $[1,2,3,4]$, найкращим є хід A_1 , а якщо присвоюються значення $[1,20,30,400]$, найкращим стає хід A_2 . Тому програма поводиться цілком по-різному, якщо вносяться зміни в шкалу деяких оціночних значень! Як виявилося, такої чутливості до зміни шкали можна уникнути за умови, що функція оцінки являє собою позитивну лінійну трансформацію ймовірності виграшу з деякої позиції (або, в більш загальному сенсі, очікуваної корисності даної позиції).



Мал. 6.10. Приклад того, що в результаті трансформації оціночних значень листя, при якій не змінюється впорядкування цих значень, найкращий хід стає іншим

Складність оцінки очікуваних мінімаксних значень

Якби в програмі були заздалегідь відомі всі випадання жереба, які повинні відбутися протягом іншої частини гри, то пошук рішення в грі з жеребкуванням був би повністю аналогічним пошуку рішення в грі без жеребкування, який здійснюється мінімаксним алгоритмом пошуку за час $O(b^m)$. Але оскільки в алгоритмі, що використовує очікувані мінімаксні значення, розглядаються також всі можливі послідовності випадання жереба, для його роботи потрібен час $O(b^m n^m)$, де n - кількість різних варіантів випадання жереба.

Навіть якщо глибина пошуку обмежена деякою невеликою величиною d , через такі додаткові витрати часу, набагато більш значні у порівнянні з

мінімаксним алгоритмом, в більшості ігор з елементами випадковості стає неможливим прагнення заглянути в дерево пошуку на дуже велику глибину. В нардах n дорівнює 21, а b зазвичай становить приблизно 20, але в деяких ситуаціях може досягати величини 4000 для випадінь жереба, що включають очки, які повторюються. Мабуть, все, на що можна розраховувати, - це зазирнути вперед на три напівходи.

Ще один спосіб роздумів про цю проблему полягає в наступному: перевагою альфа-бета-пошуку є те, що в ньому ігноруються майбутні напрямки розвитку гри, які просто не повинні бути реалізовані, якщо в грі завжди вибирається найкращий хід. Тому альфа-бета-пошук зосереджується на найбільш ймовірних подіях. В іграх з жеребкуванням ймовірних послідовностей ходів не може бути, оскільки, для того, щоб були зроблені дані ходи, спочатку повинен випасти правильний жереб, який зробив би їх допустимими. У цьому полягає загальна проблема, що виникає в будь-який такій ситуації, коли в картину світу втручається невизначеність: кількість варіантів подальших дій множиться в надзвичайному ступені, а формування докладних планів дій стає безглуздим, оскільки світ, швидше за все, буде грати зовсім в іншу гру.

Безсумнівно, читачеві вже приходила в голову думка, що до дерев ігор з вузлами жеребкування, мабуть, можна було б застосувати якийсь метод, подібний альфа-бета-відсіканню. Як виявилося, така можливість дійсно існує. Аналіз вузлів MIN і MAX залишається незмінним, але можна також забезпечити відсікання вузлів жеребкування з використанням певної частки винахідливості. Розглянемо вузол жеребкування C, наведений на [рис. 6.9](#), і визначимо, що буде відбуватися із значенням цього вузла в міру дослідження і оцінки його дочірніх вузлів. Чи можна знайти верхню межу значення C до того, як будуть перевірені всі його дочірні вузли? (Нагадаємо, що саме це потрібно в альфа-бета-пошуку для того, щоб можна було виконати відсікання вузла і його піддерева.) На перший погляд така вимога може здатися нездійсненою, оскільки значення вузла C являє собою середнє значення його дочірніх вузлів. А до тих пір, поки не будуть розглянуті всі випадання жереба, це середнє може

бути чим завгодно, оскільки самі недосліжені дочірні вузли можуть мати взагалі будь-яке значення. Проте, якщо будуть встановлені межі допустимих значень функції корисності, то з'явиться можливість визначати межі і цього середнього. Наприклад, якщо буде встановлено, що всі значення корисності знаходяться в межах від +3 до -3, то значення листових вузлів стають обмеженими, а це, в свою чергу, дозволяє встановити верхню межу значення вузла жеребкування, не розглядаючи всі його дочірні вузли.

5.6 Ігри з неповною інформацією

Байєсова гра або гра з неповною інформацією в теорії ігор характеризуються неповнотою інформації про суперників (їх можливі стратегії і виграші), при цьому у гравців є віри щодо цієї невизначеності.

Байєсову гру можна перетворити в гру повної, але недосконалої інформації, якщо прийняти допущення про загальний априорний розподіл.

На відміну від неповної інформації, недосконала інформація включає знання стратегій і виграшів суперників, але історія гри (попередні дії опонентів) доступна не всім учасникам.

Джон Харсані описав байєсові ігри наступним чином. На додаток до фактичних учасників гри з'являється віртуальний гравець «Природа». Природа наділяє кожного з фактичних учасників випадкової змінної, значення якої називаються типами. Розподіл (щільність або функція ймовірності) типів для кожного з гравців відомий. На початку гри природа «обирає» типи гравців.

Таким чином, неповнота інформації в байєсовій грі - незнання принаймні одним гравцем типу якогось іншого учасника. Гравці мають віри щодо типів суперників; віра - ймовірнісний розподіл на безлічі можливих типів. В процесі гри віри оновлюються відповідно до теореми Байєса.

Набрати мат визначення гри.

5.6.1 Карткові ігри

Карткові ігри цікаві не тільки тим, що вони часто бувають пов'язані з грошовими ставками, а й з багатьох інших причин. Кількість різних варіантів карткових ігор надзвичайно велика, але в цьому розділі ми зосередимося на таких варіантах, в яких карти тасують випадковим чином на початку гри і кожен гравець отримує на руки карти, які він не показує іншим гравцям. До таких ігор належать бридж, віст, "Черви" і деякі види покеру.

На перший погляд може здатися, що карткові ігри повністю аналогічні іграм з жеребкуванням: роздача карт відбувається випадковим чином, а самі карти визначають, які ходи можуть бути зроблені кожним гравцем. Однак в картах все жеребкування відбувається з самого початку! Це зауваження буде обговорюватися нижче більш детально і буде показано, що така особливість розглянутих карткових ігор дуже корисна на практиці. Разом з тим, це зауваження одночасно є абсолютно неправильним по дуже цікавим причинам.

Уявіть собі, що два гравці, MAX і MIN, розігрують якусь реальну роздачу по чотири карти в бриджі з двома гравцями, в якому відкриті всі карти. На руках знаходяться наступні карти, і гравець MAX повинен ходити першим:

MAX: **♥ 6 ♦ 6 ♣ 8 7** MIN: **♥ 4 ♠ 2 ♣ 9 3**

Припустимо, що гравець MAX ходить з карти **♣ 8**. Тепер повинен ходити гравець MIN, який може викинути карту **♣ 9** або **♣ 3**. Гравець MIN кладе карту **♣ 9** і забирає хабар. Потім черга ходу переходить до гравця MIN, який ходить картою **♠ 2**. У гравця MAX масти піка немає (і тому він не може забрати цей хабар), отже, він зобов'язаний викинути якусь іншу карту. Очевидним варіантом є карта **♦ 6**, оскільки дві інші карти, які залишилися є старшими. Тепер, якою б картою не ходив гравець MIN під час наступного розіграшу, гравець MAX візьме два хабара, що залишилися і гра закінчиться нічиєю, при двох хабарах у кожного гравця. З використанням відповідного варіанту мінімаксного пошуку можна легко показати, що фактично хід картою **♣ 8**, зроблений гравцем MAX, був оптимальним.

Тепер замінимо карти на руках гравця MIN і замість карти ♥ 4 введемо карту ♦ 4:

MAX: ♥ 6 ♦ 6 ♣ 8 7 MIN: ♥ 4 ♠ 2 ♣ 9 3

Два розглянуті випадки є повністю симетричними: хід гри буде однаковим, за винятком того, що при розіграші другого хабара гравець MAX викине карту ♥ 6. Гра знову закінчиться нічиєю при двох хабарах у кожного гравця, і хід картою ♣ 8 є оптимальним.

До сих пір все йшло добре. А тепер розкриємо одну з карт гравця MIN і припустимо, що гравець MAX знає, що у гравця MIN на руках або перша роздача (з картою ♥ 4), або друга роздача (з картою ♦ 4), але він не знає, яка саме з них. Гравець MAX міркує наступним чином.

Хід картою ♣ 8 є оптимальним рішенням у грі проти першої та другої роздачі на руках гравця MIN, тому тепер цей хід повинен бути оптимальним, оскільки відомо, що на руках у гравця MIN є один із цих двох варіантів роздачі.

На більш узагальненому рівні можна сказати, що гравець MAX використовує підхід, який може бути названий "усередненням за прогнозами". Ідея його полягає в тому, щоб при наявності карт на руках у противника, які не видно гравцеві, оцінювати кожен можливий варіант дій, спочатку обчислюючи мінімаксне значення цієї події стосовноожної можливої роздачі карт, а потім обчислюючи очікуване значення за всіма роздачами з використанням ймовірностіожної роздачі.

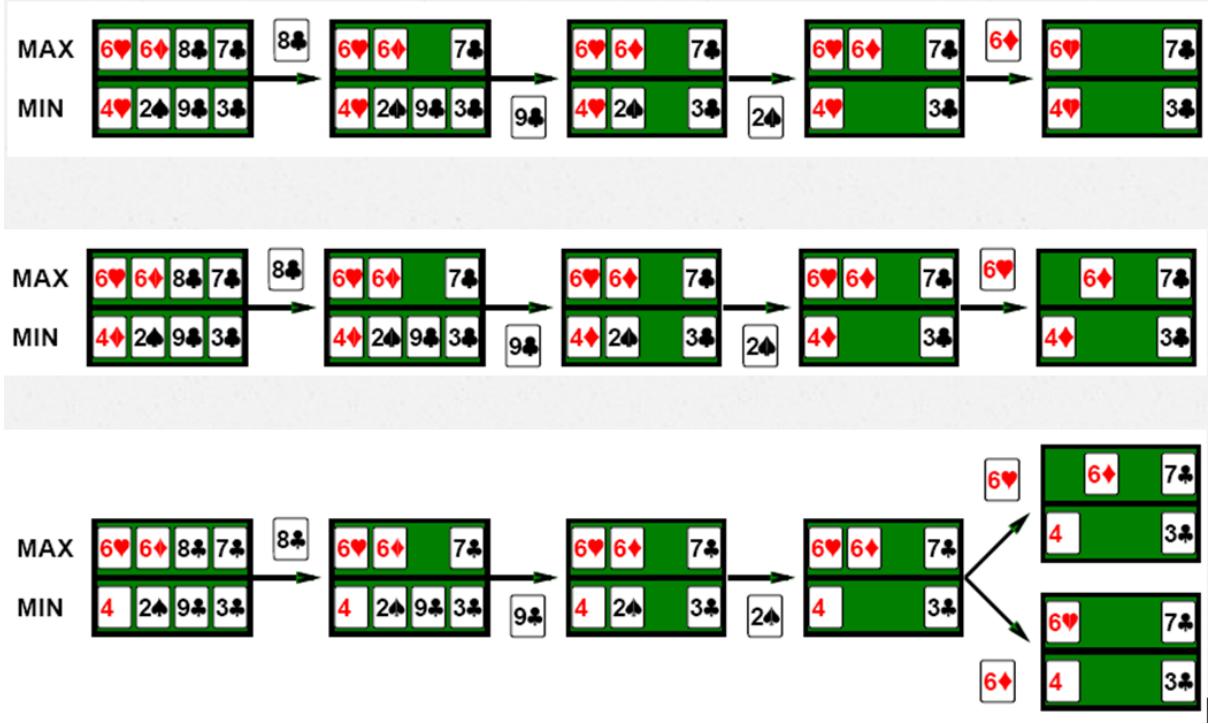
Якщо читач вважатиме такий підхід розумним (або якщо він не може судити про нього, оскільки не знайомий з бридже), то йому слід подумати над наведеною нижче розповіддю.

Перший день. Дорога A веде до купи золотих злитків; дорога B веде до розвилки. Якщо ви від розвилки підете ліворуч, то знайдете гору коштовностей, а якщо підете праворуч, то потрапите під автобус.

Другий день. Дорога А веде до купи золотих злитків; дорога В веде до розвилки. Якщо ви від розвилки підете праворуч, то знайдете гору коштовностей, а якщо підете ліворуч, то потрапите під автобус.

Третій день. Дорога А веде до купи золотих злитків; дорога В веде до розвилки. Якщо ви від розвилки виберете правильний напрямок, то знайдете гору коштовностей, а якщо неправильний, то потрапите під автобус.

Очевидно, що в перші два дні рішення вибрати дорогу В не позбавлене сенсу, але в третій день жодна людина при здоровому глузді не вибере дорогу В. Однак саме такий варіант підказує метод усереднення за прогнозами: дорога В є оптимальною в ситуаціях, що виникають в перший і другий дні, тому вона оптимальна і в третій день, оскільки повинна мати місце одна з двох попередніх ситуацій. Повернемося до карткової гри: після того як гравець MAX піде картою ♣ 8, гравець MIN забере хабар картою ♣ 9. Як і раніше, MIN піде з карти ♠ 2, але тепер гравець MAX виявиться перед розвилкою на дорозі без будь-яких вказівок. Якщо гравець MAX викине карту ♥ 6, а у гравця MIN все ще залишатиметься карта ♥ 4, то ця карта ♥ 4 стане козирною і гравець MAX програє гру. Аналогічним чином, якщо гравець MAX викине карту ♦ 6, а у гравця MIN все ще залишатиметься карта ♦ 4, гравець MAX також програє. Тому гра з першим ходом картою ♣ 8 веде до ситуації, в якій гравець MAX має 50%-ву ймовірність програшу. (Для нього було б набагато краще спочатку зіграти картами ♥ 6 і ♦ 6, гарантуючи для себе нічийну гру.)



Малюнок

З усього цього можна винести урок, що при нестачі інформації гравець повинен враховувати, яку інформацію він буде мати в кожен момент гри. Недолік алгоритму, що застосовується гравцем MAX, полягає в тому, що в ньому передбачається, ніби при кожній можливій роздачі гра буде розвиватися так, як якщо б всі карти залишалися видимими. Як показує даний приклад, це змушує гравця MAX діяти таким чином, як ніби невизначеність вирішиться, коли настане час. Крім того, в алгоритмі гравця MAX ніколи не приймається рішення, що потрібно збирати інформацію (або надавати інформацію партнеру), оскільки цього не потрібно робити в рамках кожної окремої роздачі; проте в таких іграх, як бридж, часто буває доцільно зіграти такою картою, яка допомогла б дещо дізнатися про карти противника або повідомити партнеру про свої власні картах. Такі стереотипи поведінки формуються автоматично оптимальним алгоритмом ведення ігор з неповною інформацією. Подібний алгоритм виконує пошук не в просторі станів світу (під цим маються на увазі карти, що знаходяться на руках у гравців), а в просторі **довірчих станів** (уявлень про те, хто які карти має і з якими ймовірностями). Автори зможуть пояснити цей алгоритм належним чином в главі 17 після розробки всього необхідного ймовірнісного інструментарію. А в цьому розділі необхідно також

зупинитися на одному заключному і дуже важливому зауваженні: в іграх з неповною інформацією найкраще видавати противнику як можна менше інформації, а найкращий спосіб зробити це частіше за все полягає в тому, щоб діяти непередбачувано. Ось чому санітарні лікарі, відвідуючи для перевірки підприємства громадського харчування, не повідомляють заздалегідь про свої візити.

Прийняття рішень при наявності декількох агентів: теорія ігор

В даному розділі розглядаються деякі ідеї **теорії ігор**, які можуть застосовуватися при аналізі ігор з одночасно виконуваними ходами. Для спрощення викладу спочатку розглянемо ігри, які тривають лише протягом одного ходу. На перший погляд може здатися, що слово "гра" не зовсім підходить для позначення такого спрощення, яке зводиться до одного ходу, але в дійсності теорія ігор використовується в дуже серйозних ситуаціях прийняття рішень, включаючи ведення справ про банкрутство, організацію аукціонів з розподілу спектра радіочастот, прийняття рішень по розробці промислової продукції і призначенню на неї цін, а також національну оборону. У таких ситуаціях мова часто йде про мільярди доларів і сотні тисяч людських життів. Теорія ігор може використовуватися щонайменше в двох описаних нижче напрямках.

1. Проектування агента. Теорія ігор дозволяє аналізувати рішення агента і обчислювати очікувану корисність для кожного рішення (з урахуванням припущення, що інші агенти діють оптимальним чином відповідно до теорії ігор). Наприклад, в грі в парне і непарне на двох пальцях (цю гру на пальцях називають також *morra*, від італійського слова *camorra* - група) два гравці, *O* (*Odd* - непарний) і *E* (*Even* - парний), одночасно показують один або два пальці. Припустимо, що загальна кількість показаних пальців дорівнює f . Якщо число f є непарним, гравець *O* отримує f доларів від гравця *E*, а якщо число f - парне, гравець *E* отримує f доларів від гравця *O*. Теорія ігор дозволяє визначити найкращу стратегію в грі проти раціонально діючого гравця і очікуваний виграш для кожного гравця.

2. Проектування механізму. Якщо в середовищі є багато агентів, то може існувати можливість так визначити правила дій в цьому середовищі (тобто правила гри, в яку повинні грati агенти), щоб загальний добробут усіх агентів максимізувався, якщо кожен агент приймає обґрунтоване теорією ігор рішення, максимізуючи його власну корисність. Наприклад, теорія ігор дозволяє проектувати такі протоколи для колекції маршрутизаторів трафіку Internet, щоб кожен маршрутизатор прагнув діяти в напрямку максимізації глобальної пропускної здатності. Проектування механізму може також використовуватися для створення інтелектуальних мультиагентних систем, які вирішують складні завдання в розподіленій формі без необхідності для кожного агента знати про те, яке завдання вирішується в цілому.

Будь-яка гра в теорії ігор визначається за допомогою описаних нижче компонентів.

Гравці, або агенти, які повинні приймати рішення. Найбільша увага в дослідженнях приділяється іграм з двома гравцями, хоча досить часто розглядаються також ігри з n гравцями, де $n > 2$. В цьому розділі імена гравців записуються з великої літери, наприклад Alice і Bob або O і E.

¹Гра в парне і непарне - це розважальна версія реальної гри в інспекцію. В таких іграх інспектор обирає день для інспектування (такого як ресторан або завод біологічної зброї), а оператор підприємства обирає день, в який потрібно заховати всі заборонені предмети. Якщо ці дні не збігаються, виграє інспектор, а якщо збігаються - виграє оператор підприємства.

Дії, які можуть бути обрані гравцями. В цьому розділі імена дій записуються малими буквами, наприклад one або testify. У розпорядженні гравців можуть перебувати однакові або неоднакові безлічі дій.

Матриця винагород, яка визначає корисність для кожного гравця кожної комбінації дій всіх гравців. Матриця винагород для гри парне чи непарне на двох пальцях приведена в табл. 17.1 (one - вибір дії, в якому гравець показує один палець, two - два пальця).

Таблиця 17.1. Матриця винагород для гри парне чи непарне на двох пальцях

		<i>O: one</i>	<i>O: two</i>
		<i>E=2, O=-2</i>	<i>E=-3, O=3</i>
<i>E: one</i>	<i>E: two</i>	<i>E=-3, O=3</i>	<i>E=4, O=-4</i>

Наприклад, в нижньому правому куті показано, що якщо гравець *O* обирає дію *two*, а гравець *E* також вибирає *two*, то винагорода для *E* дорівнює 4, а для *O* дорівнює -4.

Кожен гравець, який бере участь у грі, повинен виробити, а потім здійснити **стратегію** (нагадаємо, що такий самий термін використовується для позначення способу вибору дій не тільки в теорії ігор, але і в теорії прийняття рішень).

Чиста стратегія - це детермінована стратегія, яка визначає конкретну дія, яка повинна бути виконана в кожній ситуації; в грі, що складається з одного ходу, чиста стратегія складається з однієї дії. Аналіз ігор призводить до формулювання ідеї змішаної стратегії, яка являє собою рандомізовану стратегію, що передбачає вибір конкретних дій серед доступних дій відповідно до визначеного розподілу ймовірностей. **Змішана стратегія**, в якій зазвичай обирається дія *a* з ймовірністю *p*, а в іншому випадку обирається дія *b*, умовно позначається як **[p: a; (1-p) :b]**. Наприклад, змішаною стратегією для гри в парне і непарне на двох пальцях може бути **[0.5: one; 0.5:two]**. Профілем стратегії називається варіант присвоювання стратегії кожному гравцеві; після того як заданий профіль стратегії, результат гри для кожного гравця приймає певне числове значення.

Рішенням гри називається профіль стратегій, в якому кожен гравець приймає раціональну стратегію. Як буде показано нижче, найбільш важливою проблемою в теорії ігор є визначення того, що мається на увазі під словом "раціональний", коли кожен агент обирає тільки частину профілю стратегій, що визначає цей результат. Важливо зрозуміти, що результатами є фактичні

підсумки ведення гри, а рішення є теоретичними конструкціями, які використовуються для аналізу гри. Нижче буде показано, що деякі ігри мають рішення тільки в змішаних стратегіях. Але це не означає, що гравець повинен в буквальному сенсі слова приймати змішану стратегію, щоб діяти раціонально.

Розглянемо описану нижче історію. Два відчайдушних грабіжника, Аліса і Боб, були спіймані на місці злочину недалеко від місця скоєного ними пограбування, і тепер їх окремо допитують слідчі. Обидва вони знають, що, зізнавшись у спільному вчиненні цього злочину, отримають по 5 років тюремного ув'язнення за грабіж, а якщо обидва відмовляться зізнаватися, то отримають тільки по 1 року кожен за менш грубе правопорушення - володіння краденим майном. Але слідчі пропонують окремо кожному з них таку угоду: якщо ви дасте свідчення проти вашого партнера як ватажка зграї грабіжників, то вас звільнять, а партнера засудять на 10 років. Тепер Аліса і Боб стикаються з так званою **дилемою ув'язненого**: чи повинні вони свідчити (*testify*) або відмовитися (*refuse*)? Будучи раціональними агентами, і Аліса, і Боб бажають максимізувати свою власну очікувану корисність. Припустимо, що Аліса повністю байдужа до долі свого партнера, а її оцінка корисності зменшується пропорційно кількості років, які вона проведе у в'язниці, незалежно від того, що трапиться з Бобом. Боб думає так само. Щоб спростити для себе вироблення раціонального рішення, обидва грабіжники складають матрицю винагород, наведену в табл. 17.2.

Таблиця 17.2. Матриця винагород для дилеми ув'язненого

	Alice: testify	Alice: refuse
Bob: testify	$A = -5, B = -5$	$A = -10, B = 0$
Bob: refuse	$A = 0, B = -10$	$A = -1, B = -1$

Аліса аналізує цю матрицю винагород наступним чином: припустимо, що Боб буде свідчити проти мене. В такому випадку я отримаю 5 років, якщо буду свідчити проти нього, і 10 років - коли відмовлюся, тому в даному випадку краще свідчити проти нього. З іншого боку, якщо Боб відмовиться, то я отримаю 0 років, якщо буду свідчити, і 1 рік, якщо відмовлюся, тому і в даному

випадку краще свідчити. Отже, і в тому і в іншому випадку для мене краще свідчити проти Боба, тому так я і повинна вчинити.

Аліса виявила, що *testify* - це **домінантна стратегія** для гри, яку ми розглядаємо. Прийнято вважати, що стратегія s для гравця p **строго домінує** над стратегією s' , якщо результат стратегії s краще для гравця p , ніж результат стратегії s' , при будь-якому виборі стратегій іншими гравцями. Стратегія s **слабо домінує** над s' , якщо s краще ніж s' щонайменше в одному профілі стратегій і не гірше в будь-якому іншому. Домінантною стратегією є стратегія, яка домінує над усіма іншими. Нерационально вести гру на основі стратегії, над якою строго домінують інші стратегії, і нерационально не вести гру на основі домінантної стратегії, якщо вона існує. Будучи раціональним агентом, Аліса обирає домінантну стратегію. Перш ніж перейти до подальшого викладу, необхідно розглянути ще кілька термінів: прийнято вважати, що деякий результат є **оптимальним згідно з принципом Парето²**, якщо немає іншого результату, який вважали би кращим всі гравці. Над результатом **домінує за принципом Парето** інший результат, якщо всі гравці вважають кращим цей інший результат.

А якщо Аліса - не тільки раціональний, а й розумний агент, вона повинна продовжити міркування наступним чином: домінантною стратегією Боба також є засвідчення против мене. Тому він буде свідчити, і ми обидва отримаємо по п'ять років. Якщо обидва гравці мають домінантну стратегію, то комбінація цих стратегій називається **рівновагою домінантних стратегій**. Взагалі кажучи, профіль стратегій утворює **рівновагу**, якщо жоден гравець не може отримати вигоду від перемикання з однієї стратегії на іншу, за умови, що всі інші гравці продовжують дотримуватися однієї і тієї ж стратегії. Рівновага по суті являє собою **локальний оптимум** в просторі стратегій; вона є вершиною піка, навколо якої знаходяться спади уздовж всіх вимірювань, де вимірювання відповідають варіантам вибору стратегії гравцем.

Парадоксальність дилеми ув'язненого полягає в тому, що результат в точці рівноваги для обох гравців гірше в порівнянні з результатом, якого вони

досягли б, якби обидва відмовилися свідчити один проти одного. Іншими словами, результат для рівноважного рішення домінується за принципом Парето результатом (-1, -1), який відповідає обопільній відмові від свідоцтва, (*refuse, refuse*).

Чи існує який-небудь спосіб, за допомогою якого Аліса і Боб могли б формально прийти до результату (-1, -1)? Безумовно, що для них обох варіант, в якому вони відмовляються свідчити, є допустимим, але цей варіант малоймовірний.

²Оптимальність за принципом Парето названа на честь економіста Вільфредо Парето (1848-1923)

Будь-який гравець, який аналізує варіант ходу *refuse*, зрозуміє, що для нього було б краще вибрати хід *testify*. У цьому полягає притягальна сила точки рівноваги.

Математик Джон Неш (народ. в 1928 році) довів теорему, згідно з якою *кожна гра має рівновагу такого типу, яка визначена в даному прикладі*.

Тепер зазначену умову називають в його честь **рівновагою Неша**. Очевидно, що рівновагою домінантних стратегій є рівновага Неша, але не всі ігри мають домінантні стратегії. Теорема Неша означає, що рівноважні стратегії можуть існувати навіть при відсутності домінантних стратегій.

У дилемі ув'язненого рівновагою Неша є тільки профіль стратегій (*testify, testify*). Важко зрозуміти, як раціональні гравці можуть уникнути такого результату, оскільки в будь-якому запропонованому нерівноважному рішенні щонайменше один з гравців буде піддаватися спокусі змінити свою стратегію. Фахівці в області теорії ігор погоджуються з тим, що обов'язковою умовою того, щоб деякий хід був рішенням, є приналежність його до рівноваги Неша, але ще не досягнуто повне порозуміння щодо того, чи є ця умова достатньою.

Рішення (*testify, testify*) можна досить легко уникнути, трохи змінивши правила гри (або погляди гравців). Наприклад, можна перейти до ітераційної гри, в якій гравці знають, що колись обов'язково знову зустрінуться і згадають, як діяли при минулій зустрічі (але вкрай важливо те, що вони не повинні мати певну інформацію про те, через який час зустрінуться). Крім того, якщо агенти мають моральні переконання, які сприяють співпраці і справедливому ставленню один до одного, то можна змінити матрицю винагороди так, щоб вона відображала для кожного агента корисність взаємодії з іншим агентом. Як буде показано нижче, на результатах може також відбитися така модифікація агентів, щоб вони володіли обмеженою обчислювальною потужністю, а не здатністю міркувати абсолютно раціонально, а також надання одному агенту інформації про те, що здібності іншого міркувати раціонально обмежені.

Тепер розглянемо гру, в якій відсутня домінантна стратегія. Припустимо, що компанія Acme, виробник апаратних засобів для відеоігор, повинна вирішити, чи будуть в її наступній ігровій машині використовуватися DVD або CD. Тим часом компанія Best, виробник програмного забезпечення для відеоігор, повинна прийняти рішення про те, чи варто випускати свою чергову гру на DVD або CD. Для обох компаній прибуток буде позитивним, якщо вони приймуть узгоджені рішення, і негативним, якщо рішення не будуть узгодженими, як показує матриця винагород, наведена в табл. 17.3.

Таблиця 17.3. Матриця винагород для компаній Acme і Best

	<i>Acme: dvd</i>	<i>Acme: cd</i>
<i>Best: dvd</i>	$A=9, B=9$	$A=-4, B=-1$
<i>Best: cd</i>	$A=-3, B= -1$	$A=5, B=5$

Для цієї гри відсутня рівновага домінантних стратегій, але є дві рівноваги Неша: (*dvd, dvd*) і (*cd, cd*). Відомо, що це - рівноваги Неша, оскільки, якщо один з гравців прийме одностороннє рішення перейти на іншу стратегію, ситуація для нього погіршиться. Тепер ці агенти стикаються з такою проблемою: є кілька прийнятних рішень, але якщо кожен агент вибере інше рішення, то результатуючий профіль стратегій взагалі не буде являти собою рішення, і

обидва агенти зазнають збитків. Яким чином ці гравці можуть узгодженого прийти до якогось рішення? Одна з можливих відповідей полягає в тому, що обидва гравці повинні вибрати рішення, оптимальне за принципом Парето, - (dvd, dvd); це означає, що можна обмежити визначення поняття "рішення" унікальним, оптимальним за принципом Парето, рівновагою Неша, за умови, що воно існує. Кожна гра має щонайменше одне оптимальне за принципом Парето рішення, але в будь-якій грі може бути кілька таких рішень, або ці рішення можуть не виявитися точками рівноваги. Наприклад, можна було б встановити винагороди за рішення (dvd, dvd), рівні 5, а не 9. В цьому випадку є дві однакові точки рівноваги, оптимальні за принципом Парето. Щоб вибрати між ними, агенти повинні або користуватися здогадками, або вдаватися до спілкування, що можна зробити, прийнявши угоду про впорядкування рішень до початку гри, або проводити переговори, щоб досягти обопільно вигідного рішення під час гри (це може означати, що до складу багатоходової гри повинні бути включені комунікативні дії). Таким чином, необхідність в спілкуванні виникає в рамках теорії ігор точно з тих же причин, що і в мультиагентному плануванні, описаному в розділі 12. Ігри, подібні до цієї, в яких гравці повинні вступати у взаємодію, називаються **координаційними іграми**.

Вище було показано, що гра може мати більше однієї рівноваги Неша; але на підставі чого ми можемо стверджувати, що кожна гра повинна мати щонайменше одну таку рівновагу? Може виявитися, що в грі відсутня рівновага Неша для чистої (не змішаної) стратегії. Розглянемо, наприклад, будь-який профіль чистих стратегій для гри в парне і непарне на двох пальцях. Якщо загальна кількість показаних пальців є парною, то гравець O може захотіти перейти на іншу стратегію, а якщо ця кількість непарна, то буде прагнути перейти гравець E . Тому жоден профіль чистих стратегій не може являти собою рівновагу і доводиться розглядати змішані стратегії.

Але якою має бути змішана стратегія? У 1928 році фон Нейман розробив метод пошуку оптимальної змішаної стратегії для ігор з двома гравцями, які називаються **іграми з нульовою сумою**. Грою з нульовою сумою називається гра, в якій винагороди в кожній комірці матриці винагороди в сумі дорівнюють

нулю³. Очевидно, що гра в парне і непарне є саме такою. Відомо, що для ігор з двома гравцями і нульовою сумаю винагороди є рівними і протилежними, тому досить розглянути винагороди тільки для одного гравця, який буде вважатися максимізуючим гравцем ([так само, як і в розділі 6](#)).

Стосовно до гри в парне і непарне виберемо в якості максимізуючого гравця E , який вибрав для себе в якості виграшного результату парну кількість пальців, тому можна визначити матрицю винагород на основі значень

$U_E(e, o)$ - винагорода, що одержується гравцем E , якщо гравець E вибирає дію e , а O - дію o .

Метод фон Неймана називається методом **максиміна** і діє, як описано нижче.

Припустимо, що правила гри змінилися таким чином, що гравець E змушений розкривати свою стратегію першим, а за ним слідує гравець O . В такому випадку ми маємо справу з грою, де ходи виконуються по черзі, до якої можна застосувати стандартний **мінімаксний** алгоритм з глави [6](#).

Припустимо, що така гра призводить до отримання результату $U_{E, o}$. Очевидно, що ця гра закінчиться на користь гравця O , тому справжня корисність U даної гри (з точки зору гравця E) дорівнює щонайменше $U_{E, o}$.

³Більш загальним є поняття **ігор з постійною сумаю**, в яких сума в кожній комірці матриці винагород гри дорівнює деякій константі c . Будь-яку гру з постійною сумаю і n можна перетворити в гру з нульовою сумаю, віднявши з кожної винагороди значення c / n . Таким чином, шахи, в яких за традицією застосовується винагорода 1 за перемогу, $1/2$ - за нічию і 0 - за програш, формально представляє собою гру з постійною сумаю, де $c = 1$, але її можна легко перетворити в гру з нульовою сумаю, віднявши $1/2$ з кожної винагороди

Наприклад, якщо розглядаються тільки чисті стратегії, то мінімаксне дерево гри має кореневе значення, рівне -3 (рис. 17.9, а), тому відомо, що $U \geq -3$.

Тепер припустимо, що правила змінилися таким чином, що свою стратегію змушений розкривати гравець O , а за ним слідує гравець E . В такому випадку мінімаксне значення цієї гри стає рівним U_O , а оскільки гра складається на користь гравця E , то відомо, що корисність U щонайбільше дорівнює U_O . При використанні чистих стратегій це значення дорівнює +2 (див. рис. 17.9, б), тому відомо, що $U \leq +2$

Розглядаючи ці два припущення спільно, можна прийти до висновку, що справжня корисність U розглянутого рішення повинна задовольняти наступну нерівність:

$$U_{E,O} \leq U \leq U_{O,E}, \text{ або в даному випадку } -3 \leq U \leq 2$$

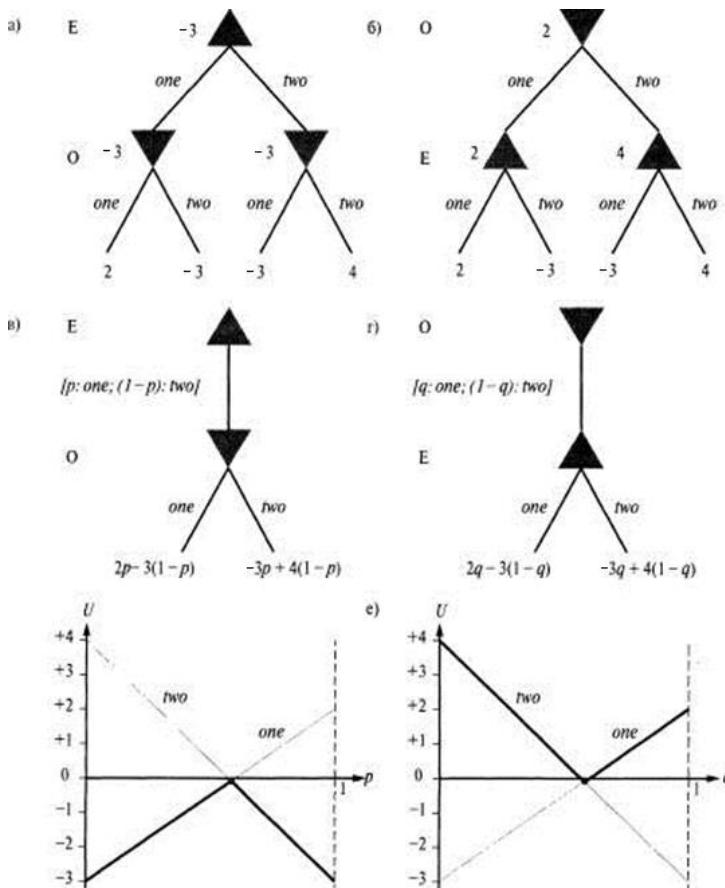
Щоб точно визначити значення U , необхідно перейти до аналізу змішаних стратегій. Спочатку зазначимо наступне: як тільки перший гравець розкрив свою стратегію, другий гравець не може програти, ведучи гру згідно чистої стратегії. Причина цього проста - якщо другий гравець веде гру на основі змішаної стратегії $[p: one; (1-p) : two]$, то очікувана корисність цієї гри являє собою лінійну комбінацію $(p \cdot U_{one} + (1-p) \cdot U_{two})$ корисностей чистих стратегій U_{one} і U_{two} . Ця лінійна комбінація ні за яких умов не буде кращою в порівнянні з кращим із значень U_{one} і U_{two} , тому другий гравець цілком може просто вибрати для ведення гри чисту стратегію.

З урахуванням цього зауваження мінімаксні дерева можна розглядати як такі, що мають нескінченну кількість гілок, що виходять від кореня, які відповідають нескінченній кількості змішаних стратегій, доступних для вибору першим гравцем. Кожна з цих гілок веде до вузла з двома гілками, відповідними чистим стратегіям для другого гравця. Ці нескінчені дерева можна зобразити як кінцеві, передбачивши один "параметризований" вибір у кореня, як описано нижче.

Ситуація, що виникає, якщо гравець E ходить першим, показана на рис. 17.9, в. Гравець E робить з кореневої позиції хід $[p: one; (1-p) : two]$, а потім гравець O обирає хід з урахуванням значення p . Якщо гравець O обирає хід *one*, то очікувана винагорода (для E) стає рівною

$2p - 3(1-p) = 5p - 3$; якщо гравець O вибирає хід *two*, то очікувана винагорода дорівнює

$-3p + 4(1-p) = 4 - 7p$. Залежності, що виражают величину цих двох винагород, можна зобразити у вигляді прямих ліній на графіку, де p змінюється від 0 до 1 вздовж осі x , як показано на рис. 17.9, д. Гравець O , що мінімізує вартість гри, повинен завжди обирати найменше значення на двох прямих лініях, як показано на цьому малюнку жирними відрізками прямих. Тому найкраще рішення, яке може прийняти гравець E , вибираючи хід, який підлягає виконанню з кореневої позиції, полягає в тому, щоб вибрати значення p , яке відповідає точці перетину і визначається наступним чином:



Мал. 17.9. Аналіз гри з двома гравцями: мінімаксні дерева гри в парне і непарне на двох пальцях, якщо гравці ходять по черзі, ведучи гру на основі чистих стратегій (а), (б); параметризовані дерева гри, в якій перший гравець використовує змішану стратегію, причому винагороди залежать від параметра ймовірності (p або q) в змішаній стратегії (в), (г); для будь-якого конкретного значення параметра ймовірності другий гравець обирає "найкраще" цих двох дій, тому значення для змішаної стратегії першого гравця задається жирними лініями; перший гравець вибирає параметр ймовірності для змішаної стратегії в точці перетину (д), (е)

$$5p - 3 = 4 - 7p \Rightarrow p = \frac{7}{12}$$

Корисність для гравця E в цей момент часу дорівнює $U_{O,E} = -\frac{1}{12}$

А, якщо першим ходить гравець O , то складається ситуація, яка показана на рис. 17.9, *г*. Гравець O з кореневої позиції робить хід $[q: one; (1-q) : two]$, після чого гравець E обирає хід з урахуванням значення q . При цьому винагороди визначаються співвідношеннями $2q - 3(1-q) = 5q - 3$ і $-3q + 4(1-q) = 4 - 7q$. Знову-таки на рис. 17.9, *е* показано, що найкращий хід, який може бути зроблений гравцем O з кореневої позиції, полягає у виборі точки перетину, наступним чином:

$$5q - 3 = 4 - 7q \Rightarrow q = \frac{7}{12}$$

Корисність для гравця E в цей момент дорівнює $U_{O,E} = -\frac{1}{12}$

Тепер відомо, що справжня корисність цієї гри знаходиться в межах від $-\frac{1}{12}$ до $-\frac{1}{12}$, тобто вона точно дорівнює $-\frac{1}{12}$! (Загальний висновок полягає в тому, що в цю гру краще грати від імені гравця O , а не E .) Крім того, справжня корисність досягається при використанні змішаної стратегії $\left[\frac{7}{12} : one; \frac{5}{12} : two\right]$, якої повинні дотримуватися обидва гравці. Така стратегія називається **максімінною рівновагою** гри і відповідає рівновазі Неша. Зверніть увагу на те, що кожна складова стратегія в рівноважній змішаній стратегії має одну і ту ж очікувану корисність. В даному випадку і хід *one*, і хід *two* мають ту ж саму очікувану корисність, $-\frac{1}{12}$, що і сама змішана стратегія.

Наведений вище результат для гри в парне і непарне на двох пальцях являє собою приклад загального результата, отриманого фон Нейманом: *кожна гра з нульовою сумою з двома гравцями має максімінну рівновагу, якщо дозволені змішані стратегії*. Крім того, кожна рівновага Неша в грі з нульовою сумою являє собою максімінну рівновагу для обох гравців. Але загальний

алгоритм пошуку максімінних рівноваг в іграх з нульовою сумаю трохи складніше в порівнянні з тим, що показано у вигляді схем на рис. 17.9, д і е.

⁴Те, що ці рівняння є такими ж як і для р, обумовлено лише збіgom; збіг виник, оскільки $U_E(\text{one}, \text{two}) = U_E(\text{two}, \text{one}) = -3$. Цей збіг також пояснює, чому оптимальна стратегія є однаковою для обох гравців

Якщо кількість можливих дій дорівнює n , то змішана стратегія являє собою точку в n -вимірному просторі і прямі лінії стають гіперплощинами. Можливо також, що над деякими чистими стратегіями для другого гравця будуть домінувати інші стратегії, так що вони перестануть бути оптимальними по відношенню до будь-якої стратегії для першого гравця.

Після видалення всіх подібних стратегій (а цю операцію може знадобитися виконати неодноразово) оптимальним варіантом ходу з кореневої позиції стає найвища (або найнижча) точка перетину гіперплощин, які залишилися. Пошук цього варіанту являє собою приклад завдання **лінійного програмування** - максимізації цільової функції з урахуванням лінійних обмежень. Такі завдання можуть бути вирішені за допомогою стандартних методів за час, який поліноміально залежить від кількості дій (а також формально і від кількості бітів, які використовуються для визначення функції винагороди).

Залишається невирішеним наступне питання: як фактично повинен діяти раціональний агент при веденні такої одноходової гри, як гра в парне і непарне? Раціональний агент прийшов логічним шляхом до висновку, що $\left[\frac{7}{12} : \text{one}; \frac{5}{12} : \text{two} \right]$ являє собою максімінну рівноважну стратегію, і виходить з припущення, що знаннями про це володіє і його раціональний противник. Агент може використовувати гральну кістку з 12 сторонами або генератор випадкових чисел для вибору випадковим чином ходу, що відповідає цій змішаній стратегії, і в цьому випадку очікувана винагорода для гравця E дорівнюватиме $-\frac{1}{12}$. В іншому випадку агент може просто вирішити зробити хід one або two. У будь-

якому випадку для гравця E очікувана винагорода залишається рівною $-\frac{1}{12}$.

Дивно те, що односторонній вибір конкретної дії не зменшує очікувану винагороду для даного гравця, але якщо інший агент матиме можливість дізнатися, що даний гравець прийняв таке одностороннє рішення, то очікувана винагорода зміниться, оскільки противник зможе відкоригувати свою стратегію відповідним чином.

Пошук рішень для кінцевих ігор з ненульовою сумою (тобто рівноваг Неша) трохи складніше. Загальний підхід полягає у використанні двох етапів. По-перше, необхідно скласти список з усіх можливих послідовностей дій, які можуть утворювати змішані стратегії. Наприклад, спочатку слід перевірити всі профілі стратегій, в яких кожен гравець виконує одну дію, потім ті, в яких кожен гравець виконує або одну, або дві дії, і т.д. Кількість таких перевірок експоненціально залежить від кількості дій, тому може застосовуватися тільки у відносно простих іграх. По-друге, для кожного профілю стратегій, включеного до списку на першому етапі, необхідно провести перевірку для визначення того, чи представляє він певну рівновагу. Таке завдання виконується шляхом вирішення ряду рівнянь і нерівностей, аналогічних тим, які використовуються у випадку з нульовою сумою. У грі з двома гравцями ці рівняння є лінійними і можуть бути вирішенні за допомогою основних методів лінійного програмування, але в разі трьох або більшої кількості гравців вони є нелінійними і завдання пошуку їх вирішення може виявитися дуже складним.

До сих пір ми розглядали тільки такі ігри, які складаються з одного ходу. Найпростішим різновидом гри, що складається з декількох ходів, є **повторювана гра**, в якій гравці знову і знову стикаються з одним і тим же вибором, але кожен раз користуються знаннями історії всіх попередніх виборів всіх гравців. Профіль стратегій для гри, яка повторюється визначає вибір дії для кожного гравця на кожному часовому інтервалі для всіх можливих історій попередніх виборів. Як і у випадку завдань MDP, винагороди визначаються адитивною функцією від часу.

Розглянемо повторювану версію пошуку вирішення дилеми ув'язненого. Чи будуть Аліса і Боб діяти спільно і відмовлятися свідчити один проти одного, знаючи про те, що їм доведеться знову зустрітися? Відповідь залежить від деталей їх угоди. Наприклад, припустимо, Аліса і Боб знають, що їм доведеться провести рівно 100 раундів гри в дилему ув'язненого. В такому випадку обидва вони знають, що 100-й раунд не буде повторюваною грою, тобто що її результат не зможе вплинути на майбутні раунди, і тому обидва вони виберуть в цьому раунді домінантну стратегію *testify*. Але як тільки буде визначено результат 100-го раунду, 99-й раунд перестане впливати на наступні раунди, тому в ньому також буде досягатися рівновага домінантної стратегії в разі вибору дій (*testify, testify*). За індукцією обидва гравці повинні вибирати дію *testify* в кожному раунді, заробивши загальний термін по 500 років тюремного ув'язнення на кожного.

Змінюючи правила взаємодії, можна отримати інші рішення. Наприклад, припустимо, що після кожного раунду існує 99% шансів, що гравці знову зустрінуться. В такому випадку очікуване число раундів все ще залишається рівним 100, але жоден з гравців не знає точно, який раунд буде останнім. При таких умовах можлива поведінка, що характеризується більшим ступенем співробітництва. Наприклад, однією зі стратегій рівноваги для кожного гравця є вибір дії *refuse*, якщо інший гравець ніколи не вибирав дію *testify*. Таку стратегію можна назвати вічною карою. Припустимо, що обидва гравці прийняли дану стратегію і це відомо їм обоим. У такому випадку, за умови, що жоден з гравців не вибере дію *testify*, в будь-який момент часу очікувана сумарна винагороду в майбутньому для кожного гравця становить наступне:

$$\sum_{t=0}^{\infty} 0,99^t \cdot (-1) = -100$$

Гравець, який вибере *testify*, отримає 0 очок замість -1 в кожному наступному ході, але його сумарна очікувана майбутня винагорода стає рівною:

$$0 + \sum_{t=0}^{\infty} 0,99^t \cdot (-10) = -99$$

Тому на кожному етапі гравці не мають стимулу, який змусив би їх відмовитися від стратегії (*refuse, refuse*). Вічне покарання - це стратегія "взаємно гарантованого знищення" в рамках дилеми ув'язненого: як тільки один з гравців вирішить виконати дію *testify*, він забезпечить отримання обома гравцями великих неприємностей. Але така перспектива розвитку подій може стати застереженням, тільки якщо інший гравець знає, що ви прийняли цю стратегію або щонайменше, що ви могли її прийняти.

Існують і інші стратегії в цій грі, які є не такими безкомпромісними. Найбільш відома з них, звана **відплатою "зуб за зуб"**, передбачає, що гравець починає з дії *refuse*, а потім повторює попередній хід іншого гравця в усіх наступних ходах. Тому Аліса повинна відмовлятися свідчити проти Боба до тих пір, поки Боб відмовляється свідчити проти неї, потім повинна вибирати в своєму ході надання свідчень проти Боба, як тільки Боб дасть свідчення проти неї, але повинна повернутися до відмови від дачі показань, після того як це зробить Боб. Хоча ця стратегія дуже проста, виявилося, що вона є надзвичайно надійною та ефективною в протидії найрізноманітнішим стратегіям.

Крім того, інші рішення можуть бути отримані в результаті модифікації самих агентів, а не зміни правил їх взаємодії. Припустимо, що агенти являють собою кінцеві автомати з n станами і грають в гру із загальною кількістю ходів $m > n$. Тому агенти в якийсь момент стають нездатними представити цілий ряд ходів, які залишилися і повинні розглядати їх як невідомі. Це означає, що вони не можуть виконувати логічний висновок по індукції і мають право переходити до найбільш сприятливої рівноваги (*refuse, refuse*). В такому випадку дурість йде на користь або, скоріше, йде на користь думка супротивника про те, що ви дурні. Ваш успіх в таких повторюваних іграх залежить від думки про вас іншого гравця як про тупицю або простака, а не від ваших фактичних характеристик.

Повторювані ігри в частково спостережуваних варіантах середовища називаються іграми з **частковою інформацією**. До прикладів таких ігор відносяться карткові ігри на зразок покеру і бриджу, в яких кожен гравець

бачить тільки деяку підмножину карт, а також більш серйозні "ігри", такі як моделювання атомної війни, коли жодна зі сторін не знає місцезнаходження всіх пускових установок супротивника. Рішення ігор з частковою інформацією можна знайти, розглядаючи дерево довірчих станів, як і у випадку завдань POMDP. Одна важлива відмінність між ними полягає в тому, що власне довірчий стан є спостережуваним, а довірчий стан противника - ні. Для таких ігор практично застосувані алгоритми були розроблені тільки недавно. Наприклад, знайдено рішення для деякої спрощеної версії покеру і доведено, що варіант, в якому гравці блефують, дійсно є раціональним, принаймні в складі ретельно збалансованої змішаної стратегії. В результаті цих досліджень було зроблено одне важливе відкриття, яке полягає в тому, що змішані стратегії корисні не тільки для того, щоб зробити дії гравця непередбачуваними, але і для мінімізації обсягу інформації, яку противник може отримати зі спостережень за діями цього гравця. Щікаво відзначити, що розробники програм для гри в бридж добре розуміють важливість збору та приховування інформації, але жоден з них не запропонував використовувати рандомізовані стратегії.

До сих пір виявлялися певні бар'єри, які перешкоджали широкому використанню теорії ігор в проектах агентів. По-перше, слід зазначити, що при пошуку рішення на основі рівноваги Неша гравець припускає, що його противник з усією визначеністю буде вести гру на основі рівноважної стратегії. Це означає, що гравець не здатний врахувати будь-які переконання, які у нього можуть бути відносно того, як швидше за все будуть діяти інші гравці, і тому не зможе скористатися своїми важливими перевагами, захищаючись від загроз, які так ніколи і не матеріалізуються. Ця проблема частково вирішується завдяки використанню поняття **рівноваги Байєса — Неша**, тобто рівноваги стосовно розподілу априорних ймовірностей гравця по відношенню до стратегій інших гравців; іншими словами, рівновага **Байєса — Неша** висловлює впевненість гравця в тому, які ймовірні стратегії будуть застосовувати інші гравці. По-друге, в даний час відсутній зручний спосіб спільногого застосування стратегій управління, заснованих на теорії гри і моделі POMDP. Через

наявність цих та інших проблем теорія ігор в основному використовувалася для аналізу варіантів середовища, що знаходяться в стані рівноваги, а не для управління агентами, діючими в деякому середовищі. Але нижче буде показано, що теорія ігор може допомогти і при проектуванні варіантів середовища.

Штучний інтелект для гри в «Дурень»

Карткові ігри (і «Дурень» в тому числі) відносяться до категорії ігор з неповною інформацією. Алгоритми, що застосовуються в шахових програмах або програмах для гри в Го до них не підходять. У всякому разі, в початковому вигляді.

Збір даних

Запорукою перемоги є розвідка! А основа розвідки - ретельний аналіз повністю відкритих даних. Що нам відомо про карти противника, коли ми граємо в «Дурня»? Здавалося б трохи, адже карти від нас закриті? На справі, все не так погано. Наприклад, ми знаємо які карти у нас на руках і номінал картки, яка показує козир (якщо гра йде з козиром). Це само по собі вже непогано, але ще краще наше знання про те, що цих карт немає у противника (це в разі, якщо гра йде однією колодою). Спробуємо скласти повний список джерел даних про карти супротивників, стосовно «Дурня»:

1. Відкриті карти.
2. Карти взяті гравцями в ході гри (в тому числі ті, якими вони намагалися неуспішно відбитися).
3. Карти вийшли з гри, в результаті віdboю.
4. Карти, побити які не вдалося.

Добре, ми зібрали дані і, в якійсь мірі, знаємо, що за карти на руках у супротивника. Ми можемо обчислити ймовірність того, що противник може успішно відбитися від нашого візиту (це не означає, що він буде це робити), але цього мало! Ми все ще не можемо «взяти і просто використовувати»

мінімаксний алгоритм з альфа-бета відсіканням. Необхідна кількісна оцінка позиції.

Тож почнемо. Очевидно, що в дурні ніж старше карта, тим вигідніше мати її в руці. Тому, побудуємо алгоритм на класичній оцінці сили руки і прийнятті рішення (наприклад, про підкидання тієї чи іншої карти) на основі цієї оцінки. Припишемо карткам значення, наприклад так:

- **туз (A)** — +600 очок,
- **король (K)** — +500,
- **дама (Q)** — +400,
- **валет (J)** — +300,
- **десятка (10)** — +200,
- **дев'ятка (9)** — +100,
- **вісімка (8)** — 0,
- **сімка (7)** — -100,
- **шістка (6)** — -200,
- **п'ятірка (5)** — -300,
- **четвірка (4)** — -400,
- **трійка (3)** — -500,
- і нарешті, **двійка (2)** — -600 очок.

Козирні карти цінніше будь-яких простих (навіть козирна двійка б'є "звичайного" туза), а ієрархія в козирній масті та ж сама, тому для їх оцінки просто додамо 1300 до "базової" величини - тоді, наприклад, козирна двійка буде "коштувати" $-600 + 1300 = 700$ очок (тобто, як раз трохи більше, ніж некозирний туз).

Важливо також врахувати наступні правила оцінки:

Вигідно мати багато карт однієї гідності - не тільки тому, що ними можна "зavalити" суперника, але і з легкістю відбити атаку (особливо, якщо карти

високої гідності). Наприклад, в кінці гри рука (для простоти покладемо, що тут і далі козирі - бубни)



Багато карт однієї (звичайно ж, некозирної) масти, навпаки, мати невигідно - вони будуть "заважати" один одному. Наприклад, рука



дуже невдала - навіть якщо суперник не "виб'є" у вас першим ходом козиря і піде картою пікової масти, то всі інші підкинуті карти будуть інших мастей, і на них доведеться віддавати козирі. Крім того, з великою ймовірністю залишиться незатребуваною п'ятірка пік - всі козирі у вас гідностю вище п'ятірки, тому ні в якому разі (якщо, звичайно ж, з самого початку не зайшли картою молодше) вам не вдасться покрити нею якусь іншу карту - ймовірність взяти дуже висока.

З іншого боку, замінimo валета пік десяткою треф, а козирну шістку - трійкою:



Незважаючи на те, що ми замінили карти на більш молодші, така рука значно краще — по-перше, на трефову масть не доведеться віддавати козиря (і можна буде з більшою ймовірністю використовувати пікового туза), а по-друге, якщо ви поб'єте якусь карту вашою козирною трійкою, є ймовірність того, що хтось кине вам трійку пік (бо особливого сенсу тримати таку карту, як правило, немає), і ви скинете п'ятірку.

Разом варто нарахувати бонусні коефіцієнти залежно від кількості карт однієї гідності - наприклад, якщо немає ні однієї карти якоїсь гідності або вона тільки одна, бонуси не нараховуються, а за всі 4 карти коефіцієнт дорівнює 1.25

0.0, 0.0, 0.5, 0.75, 1.25

Необхідно нарахувати **Штрафи** за незбалансовану по маstry руку.

Врахуємо ще таку банальну річ, як кількість карт в руці. Справді, мати на початку гри 12 хороших карт дуже навіть непогано (тим більше, що кинути все одно зможуть не більше 6), а ось в кінці гри, коли крім вас залишився тільки суперник з 2 картами, це зовсім не так, тому рахуємо кількість карт, які залишилися в грі (в колоді і на руках у гравців).

Стратегія

Який стратегії слід дотримуватися при грі в «Дурня»? В першу чергу, це залежить від фази гри. У найбільш поширеному варіанті, можна виділити три основні фази:

Початкова

Проміжна

Фінальна

Перша фаза найтривалиша. Поки колода не розібрана, основне завдання гравця (крім збору даних) - накопичення на руках старших карт (особливо козирів). Це означає, що слід ходити так, щоб позбутися від якомога більшої кількості молодших карт (для того, щоб взяти відповідну кількість карт з колоди). Противників слід від колоди відтісняти, стежачи за тим, щоб вони не відчували браку в картах на руках.

Якщо на руках велика кількість старших карт, їх слід економити. Часто вигідніше взяти карти (тим самим заощадивши свої старші карти), навіть якщо є можливість відбитися. Взяті карти можна буде згодом віддати або підкинути (особливо якщо взята карта приходить в пару або трійку до карт іншої масті), а з урахуванням того, що старші карти вже у нас на руках, ймовірність того, що противники знайдуть в колоді щось цінне, мала.

У якийсь момент карт в колоді залишається так мало, що вони будуть повністю вибрані перед наступним ходом. Цю скромину фазу гри складно помітити людині, але вона може бути дуже важлива. В першу чергу, на цьому ходу, слід дбати про те, щоб забрати собі не повний комплект карт. Друга мета - відкрита карта козиря знизу колоди (якщо там лежить туз або король, за нього

варто поборотися). Крім відкритого козиря, метою боротьби можуть бути і останні карти колоди. Про них відомо менше, але в деяких випадках, можна з високою ймовірністю передбачити там наявність великих карт.

З усього сказаного випливає, що AI слід постаратися, щоб хід, після якого вся колода розбирається, припадав на його відбій (той, хто відбивався бере карти останнім), але при цьому в колоді залишалося б достатня кількість карт, щоб встигнути взяти козиря. Якщо карт в колоді мало, AI може бути вигідно бути одним з тих, хто підкидає (тим, хто забере останні карти). Це головоломне завдання, але для комп'ютера цілком посильне, оскільки точне число карт, що залишилися в колоді, йому відоме в будь-який момент гри.

Коли колода розібрана і карта, що показує козир, взята, настає фінальна стадія гри. Єдина мета гравця, в цій фазі гри — якомога швидше позбутися від усіх своїх карт. Приватним (але дуже важливим) випадком виявляється ситуація, коли гравців залишається двоє. В цьому випадку, карти на руках противника точно відомі і можна розпланувати послідовність своїх ходів так, щоб противник не міг відбитися ні на якому ходу крім останнього. Якщо це неможливо, слід відбиватися таким чином, щоб противник не мав можливості підкидати свої карти.

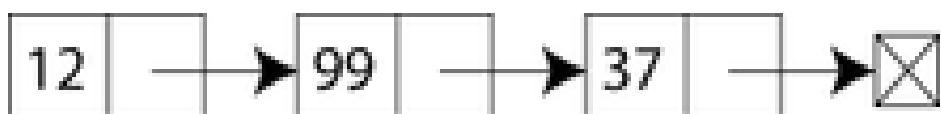
6 ДОДАТОК 1 – СПИСКОВІ СТРУКТУРИ ДАНИХ

Структура даних – це спосіб організації і зберігання сукупності елементів даних.

В інформації, список (List) - це абстрактний тип даних, що представляє собою упорядкований набір значень, в якому деяке значення може зустрічатися більше одного разу.

Екземпляром списку є комп'ютерна реалізація математичного поняття кінцевої послідовності.

Екземпляри значень, що знаходяться в списку, називаються елементами списку (Item, entry або element); якщо значення зустрічається кілька разів, кожне входження вважається окремим елементом.



Терміном список також називається кілька конкретних структур даних, що застосовуються при реалізації абстрактних списків, особливо зв'язкових списків.

Абстрактний тип даних (АТД) - це математична модель для типів даних, де тип даних визначається поведінкою (семантикою) з точки зору користувача даних, а саме в термінах можливих значень, можливих операцій над даними цього типу і поведінки цих операцій.



- ✓ ■ мають змінну довжину і прості способи її зміни
■ зміна довжини структури відбувається у визначених межах, не перевищуючи деякого максимального (границочного) значення



✓ розмір, взаєморозташування і характер зв'язків між елементами можуть значно змінюватися в процесі виконання програми

Типові операції зі структурами даних:

- пошук даних;
- зміна даних;
- додавання /видалення даних.

Спискові структури даних

- стек
- черга
- двостороння черга (дек)

- черга з приоритетом
- зв'язний список
 - однозв'язний
 - двозв'язний
 - кільцевий
 - ❖ однозв'язний
 - ❖ двозв'язний

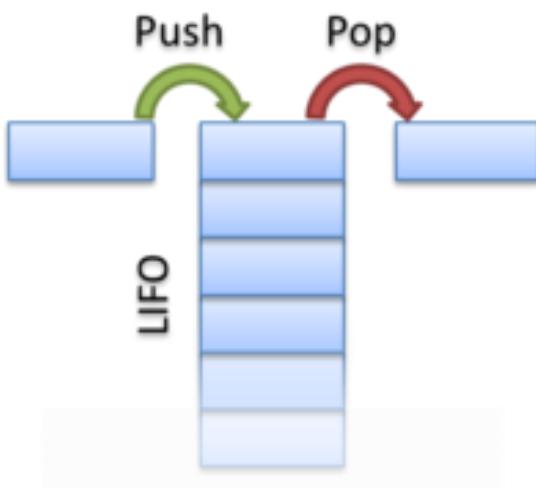
6.1 Стек

Стек - структура даних, що являє собою упорядкований набір елементів, в якій додавання нових елементів і видалення існуючих проводиться з одного кінця -вершини стека.

Притому першим з стека видаляється елемент, який був поміщений туди останнім, тобто в стеці реалізується стратегія «останнім увійшов - первім вийшов» (last-in, first-out - LIFO).

Операції стека:

- empty - перевірка стека на наявність в ньому елементів,
- push (Запис в стек) - операція вставки нового елемента,
- pop (Зняття з стека) - операція видалення останнього елемента.



Способи реалізації:

- масив
- зв'язний список
- стек-контейнер (stack)

6.2 Черга

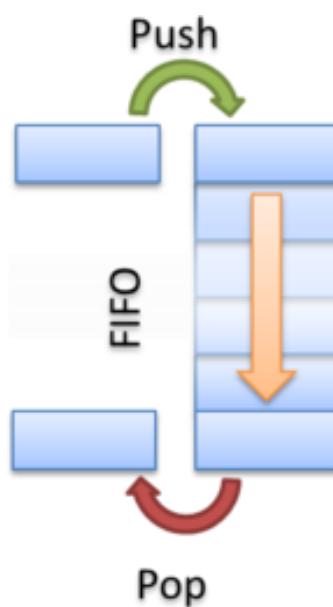
Черга - це структура даних, додавання і видалення елементів в якій відбувається шляхом операцій push і pop відповідно.

Притому першим з черги видаляється елемент, який був поміщений туди першим, тобто в черзі реалізується принцип «першим увійшов - першим вийшов» (First-in, first-out - FIFO). У черзі є голова (Head) і хвіст (Tail). Коли елемент ставиться в чергу, він займає місце в її хвості.

З черги завжди виводиться елемент, який знаходиться в її голові.

Операції черги:

- empty - перевірка черги на наявність в ній елементів,
- push (Запис в чергу) - операція вставки нового елемента,
- pop (Зняття з черги) - операція видалення нового елемента,
- size - операція отримання кількості елементів в черзі.



Способи реалізацій:

- масив

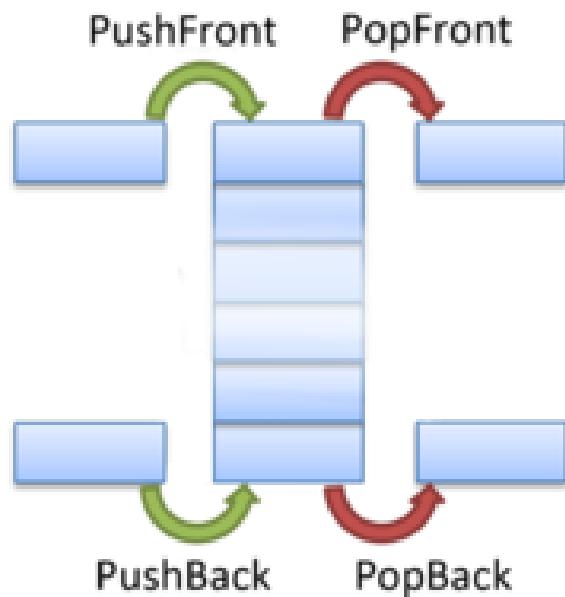
- кільцевий масив
- зв'язний список
- черга-контейнер (Deque)
- на двох стеках

6.3 Двостороння черга

Двостороння черга (жарг. ДЕК від англ. Deque - double ended queue; черга з двома кінцями) - абстрактний тип даних, в якому елементи можна додавати і видаляти як з початку, так і з кінця.

Операції двосторонньої черги:

- PushBack - додавання в кінець черги.
- PushFront - додавання на початок черги.
- PopBack - вибірка з кінця черги.
- PopFront - вибірка з початку черги.
- IsEmpty - перевірка наявності елементів.
- Clear - очищення.



Способи реалізації:

- кільцевий масив
- двозв'язний список
- дек-контейнер (queue)

6.4 Черга з пріоритетами

Черга з пріоритетами (*priority queue*) — це структура даних, що призначена для обслуговування множини елементів, кожний з яких додатково має "пріоритет", пов'язаний з ним.

У пріоритетній черзі першим обслуговується елемент, який має найвищий пріоритет, відповідно елемент, що має найнижчий пріоритет буде обслугований останнім. У деяких реалізаціях, якщо два елементи мають одинаковий пріоритет, вони подаються відповідно до порядку, в якому вони були закладені, в той час як в інших реалізаціях упорядкування елементів з одинаковим пріоритетом не визначено.

Операції черги з пріоритетами :

- `findMin` або `findMax` - пошук елемента з найбільшим (найменшим) пріоритетом,
- `insert` або `push` - вставка нового елемента,
- `extractMin` або `extractMax` – отримати елемент з найбільшим (найменшим) пріоритетом,
- `deleteMin` або `deleteMax` - видалити елемент з найбільшим (найменшим) пріоритетом,
- `increaseKey` або `decreaseKey` - оновити значення елемента,
- `merge` - об'єднання двох пріоритетних черг, зберігаючи оригінальні черги,
- `meld` - об'єднання двох пріоритетних черг, руйнуючи оригінальні черги,
- `split` - розбити пріорітну чергу на дві частини.

Способи реалізацій:

Найвна. Ми можемо взяти звичайний список і при додаванні нового елемента класти його в кінець, а при запиті елемента з максимальним пріоритетом проходити по всьому списку. Тоді операція `insert` буде виконуватися за $O(1)$, а `extractMin` або `extractMax` за $O(n)$.

За допомогою піраміди (двійкової купи)

контейнер (priority_queue)

Пріоритетні черги використовуються в наступних алгоритмах: алгоритм Дейкстри, алгоритм Прима, алгоритм Хаффмана, пошук по першому найкращому співпадінню, управління смugoю пропускання.

6.5 Зв'язаний список

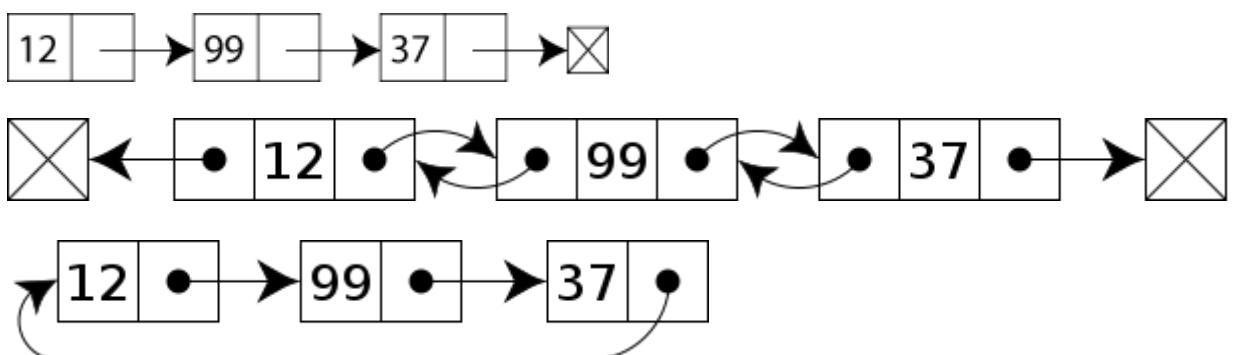
Зв'язаний список - структура даних, в якій елементи лінійно впорядковані, але порядок визначається не номерами елементів, а вказівниками, які входять до складу елементів списку та вказують на наступний за даним елемент (в однозв'язаних або однобічно зв'язаних списках) або на наступний та попередній елементи (в двозв'язаних або двобічно зв'язаних списках).

Список має «голову» — перший елемент та «хвіст» — останній елемент.

Зв'язані списки мають серію переваг порівняно з масивами.

Зокрема, в них набагато ефективніше (за час $O(1)$, тобто незалежно від кількості елементів) виконуються процедури додавання та вилучення елементів.

Натомість, масиви набагато кращі в операціях, які потребують безпосереднього доступу до кожного елементу, що у випадку зі зв'язаними списками неможливо та потребує послідовного перебору усіх елементів, які передують даному.



Способи реалізації:

В однобічно зв'язаному списку, який є найпростішим різновидом зв'язаних списків, кожний елемент складається з двох полів: *data* або даних, та вказівника *next* на наступний елемент. Якщо вказівник не вказує на інший елемент (інакше: *next = NULL*), то вважається, що даний елемент — останній в списку.

В двобічно зв'язаному списку елемент складається з трьох полів — вказівника на попередній елемент *prev*, поля даних *data* та вказівника *next* на наступний елемент. Якщо *prev=NULL*, то в елемента немає попередника (тобто він є «головою» списку), якщо *next=NULL*, то в нього немає наступника («хвіст» списка).

В кільцевому списку перший та останній елемент зв'язані. Тобто, поле *prev* голови списка вказує на хвіст списка, а поле *next* хвоста списка вказує на голову списка (для двобічно зв'язаного кільцевого списку).

```
list
{
    int field;
    list *next;
    list *prev;
};
```

Операції над зв'язаними списками:

- insert або push - вставка нового елемента,
- search – пошук елемента,
- delete – видалення елемента,
- обернення списку,
- пошук циклу .

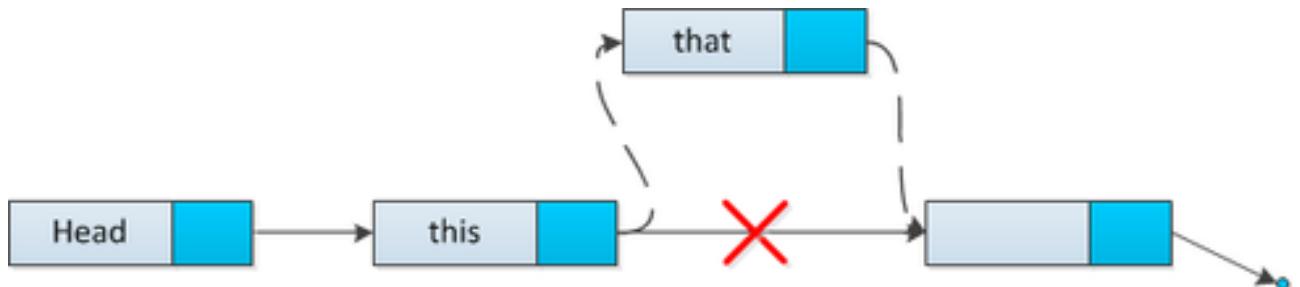
6.5.1 insert або push - вставка нового елемента,

function insert(Node thatElement):

thatElement.next = thisElement.next

thisElement.next = thatElement

Очевидним є випадок, коли необхідно додати елемент (newHead) в голову списку. Встановимо в цьому елементі посилання на стару голову, та й відновимо покажчик на голову.



6.5.2 search – пошук елемента,

Node search(**int** value):

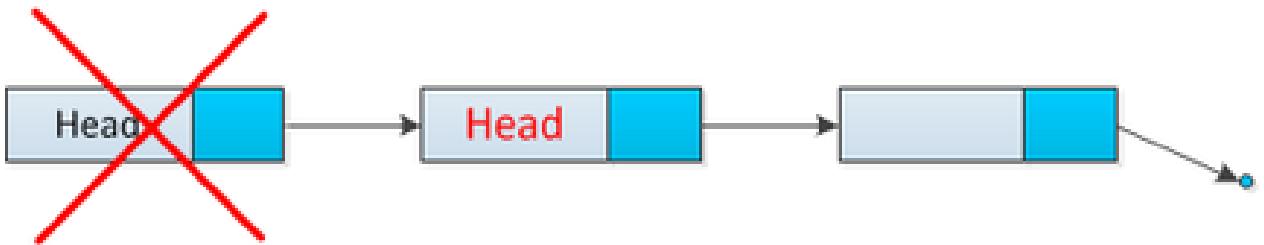
```
node = head  
while node != NULL and value != node.value  
    node = node.next  
return node
```

Для того, щоб знайти елемент зі значенням (value), будемо рухатися по списку від голови до кінця і порівнювати значення в елементах з шуканим. Якщо елементу в списку немає, то повертаємо NULL.

6.5.3 delete – видалення елемента,

Для того, щоб видалити голову списку, перепризначимо покажчик на голову на другий елемент списку, а голову видалимо.

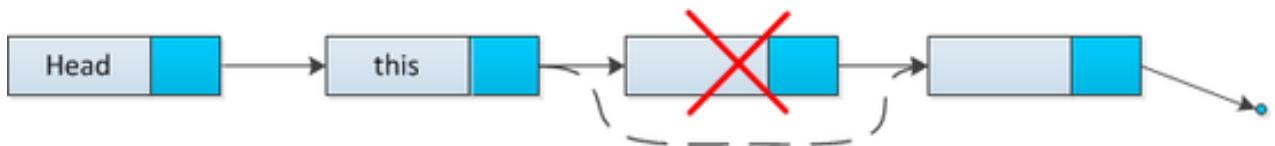
```
function removeHead():  
  
if head != NULL  
    tmp = head  
    head = head.next  
  
delete tmp
```



Видалення елемента після заданого (thisElement) відбувається наступним чином: змінимо посилання на наступний елемент на наступний за тим, що видаляється, потім видалимо потрібний елемент .

```
function removeAfter(Node thisElement):
```

```
    if thisElement.next != NULL
        tmp = thisElement.next
        thisElement.next = thisElement.next.next
        delete tmp
```



6.5.4 обернення списку,

Для того, щоб розвернути список, необхідно пройти по всіх елементах цього списку, і всі покажчики на наступний елемент замінити на попередній. Ця рекурсивна функція приймає покажчик на голову списку і попередній елемент (при запуску вказувати NULL), а повертає покажчик на нову голову списку.

```
Node reverse(Node current, Node prev):
```

```
    if current == NULL
        return prev
    next = current.next
    current.next = prev
    return reverse(next, current)
```

6.5.5 пошук циклу .

Скористаємося алгоритмом Флойда "Черепаха і заєць". Нехай за одну ітерацію перший покажчик (черепаха) переходить до наступного елементу списка, а другий покажчик (заєць) на два елементи вперед. Тоді, якщо ці два покажчика зустрінуться, то цикл знайдений, якщо дійшли до кінця списку, то циклу немає.

```
boolean hasCycle(Node head):
    tortoise = head
    hare = head
    repeat
        if hare == NULL or hare.next == NULL
            return false
        tortoise = tortoise.next
        hare = hare.next.next
    until tortoise == hare
    return true
```

Якщо циклу не існує, то заєць першим дійде до кінця і функція поверне *false*. В іншому випадку, в той момент, коли і черепаха і заєць знаходяться в циклі, відстань між ними буде скорочуватися на 1, що гарантує їх зустріч за кінцевий час.

7 ДОДАТОК 2 – ДЕРЕВОВИДНІ СТРУКТУРИ ДАНИХ

Зв'язні списки не охоплюють весь спектр можливих представлень даних. Наприклад, за їх допомогою важко описати ієрархічні структури подібні каталогам і файлам або структури для зберігання інформації генеалогічного дерева. Для цього краще підходить модель відома як дерево.

Дерева як структури даних використовуються в багатьох областях інформатики, включаючи операційні системи, графіку, бази даних, комп'ютерні мережі.

7.1 Основні визначення

Деревом називають структуру даних, кожен елемент якої позв'язний з декількома іншими її елементами. Елементи дерева називають його *вузлами* (або *вершинами*).

У спискових структурах за поточною вершиною (якщо вона не остання) завжди слідує тільки одна вершина, тоді як у деревовидних структурах таких вершин може бути декілька. Тому, на відміну від спискових структур, дерева відносяться до *нелінійних структур даних*.

Класифікацію дерев проводять за різними ознаками. Так, за кількістю можливих нащадків у вершин розрізняють *двійкові* (бінарні) та *недвійкові* дерева. *Бінарне* дерево - це дерево, кожна вершина якого має не більше двох піддерев. Якщо бінарне дерево має точно по два нащадки з кожного вузла, воно називається *повним бінарним деревом*.

Якщо в дереві важливий порядок слідування нащадків, то такі дерева називають *впорядкованими*. Для них вводиться поняття лівий і правий нащадок (для двійкових дерев) або більше лівий / правий (для недвійковий дерев). Дерево вважається впорядкованим, якщо порядок його елементів є фіксованим.

Збалансоване дерево - різновид *бінарного дерева*, яке автоматично підтримує свою висоту, тобто кількість рівнів вершин під коренем, мінімальною. Процедура зменшення (балансування) висоти дерева виконується

за допомогою трансформацій, відомих як *обернення дерева*, в певні моменти часу (переважно при видаленні або додаванні нових елементів).

Бінарне дерево називають *ідеально збалансованим*, якщо для кожної його вершини кількість вершин у лівому і правому піддереві різнятися не більше ніж на 1. Однак, така умова доволі складна для виконання на практиці і може вимагати значної перебудови дерева при додаванні або видаленні елементів. Тому існує менш строге визначення, яке отримало назву умови АВЛ-*збалансованості*: якщо висоти лівого та правого піддерев різнятися не більше ніж на 1. Дерева, що задовольняють таким умовам, називають АВЛ-деревами. Зрозуміло, що кожне ідеально збалансоване дерево є також АВЛ-збалансованим, але не навпаки.

Покроковий перебір елементів дерева по зв'язках між вузлами-предками і вузлами-нащадками називається *обходом дерева*. Оскільки дерево є нелінійної структурою, то не існує єдиної схеми обходу дерева. Класично виділяють наступні основні схеми:

- обхід *в прямому порядку* (обхід зверху вниз), при якому кожен вузол-предок переглядається перед його нащадками (рис. 16, а);
- обхід *у зворотному порядку* (обхід знизу вверх), при якому проглядаються спочатку нащадки, а потім предки ((рис. 16, б));
- *симетричний обхід* (обхід зліва направо), при якому відвідується спочатку ліве піддерево, потім вузол, потім - праве піддерево (рис. 16, в).

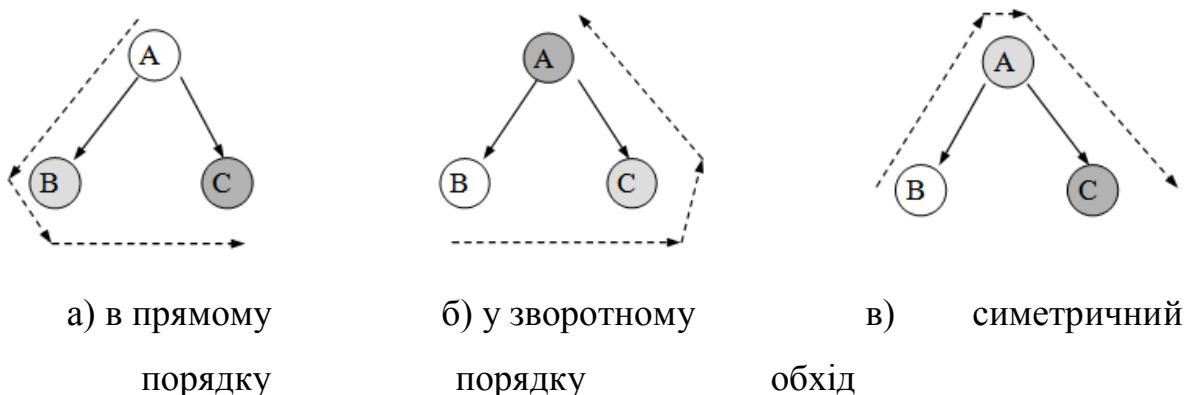
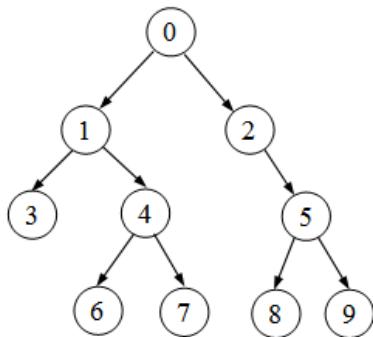


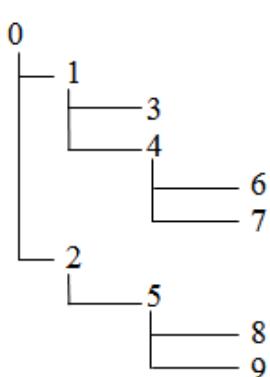
Рис. 16. Схеми обходу дерева

Іноді використовуються різновиди трьох основних правил, наприклад - обхід в обернено-симетричному порядку: праве піддерево - корінь - ліве піддерево.

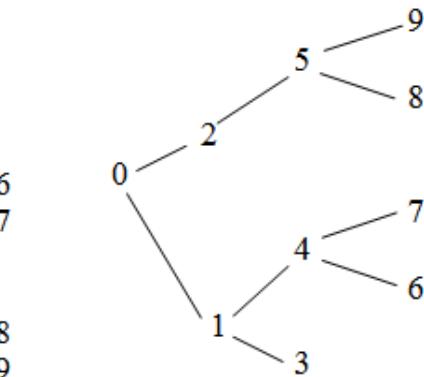
Різні правила обходу часто використовуються для виведення структури дерева в наочному графічному вигляді. Наприклад, для бінарного дерева



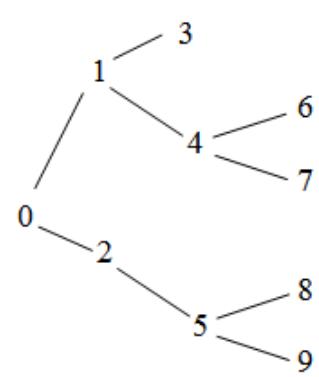
застосування різних правил обходу дозволяє отримати наступні його представлення:



Прямий обхід



Зворотно-симетричний обхід



Симетричний обхід

З цих прикладів видно, що наявність декількох правил обходу дерева цілком обґрунтована, і в кожній ситуації треба вибирати підходяче правило.

Також зручно використовувати різні правил обходу при аналізі виразів. У результаті обходу дерева зверху вниз утворюється *префіксна* форма виразу, при обході знизу вверх - *постфіксна* форма, а при обході зліва направо - *інфіксна* форма.

Найбільший практичний інтерес представляють двійкові дерева.

7.2 Бінарні дерева

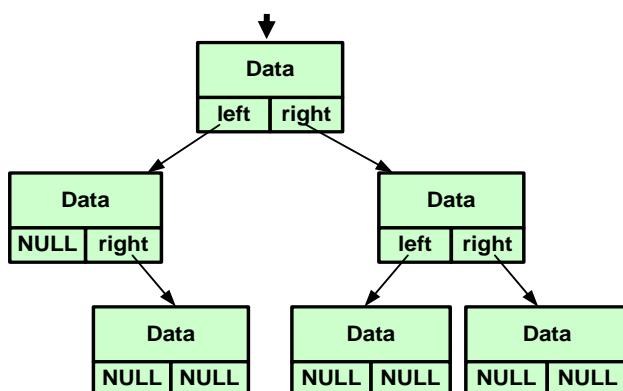
При розгляді дерева як структури даних необхідно розуміти наступне:

1) всі вершини дерева, що розглядаються як змінні мови програмування, повинні бути одного і того ж типу, більше того - записами з деяким інформаційним наповненням і необхідною кількістю сполучаючих полів;

2) в силу природної логічної розгалуженості дерев і відсутності єдиного правила розміщення вершин в порядку один за одним, їх логічна організація не співпадає з фізичним розміщенням вершин дерева в RAM.

Основний спосіб представлення дерев використовує вузли і посилання.

Для роботи з деревом, як правило, описують клас, чиїми атрибутами (явними або агрегованими) є кореневе значення і посилання (показчики) на ліве та праве піддерева. Використовуючи вузли та посилання, дерево можна представити як наступну структуру:



У C++ дерево може бути описане наступним чином:

```
class Tree

{ Node *root;      // корінь дерева

public:

    BinaryTree(): root(NULL) {}

    void insertNode(int);

    ...

};
```

Тут Node - вузол дерева, який може мати наступне оголошення:

```
struct Node

{ int data;

    Node *left, *right;
```

```
Node(int);
```

```
};
```

У цьому оголошенні атрибути `left` і `right` є посиланнями на інші сутності типу `Node`.

У Python можливе наступне оголошення класу дерева:

```
class BinaryTree:
```

```
    def __init__(self, value):
```

```
        self.key = value
```

```
        self.left = None
```

```
        self.right = None
```

```
    def insert(self, value):
```

```
    ...
```

Конструктор такого класу очікує отримати об'єкт якогось виду (`value`), щоб зберегти його в корені. Коли виконується вставка нового нащадка в дерево, то це приводить до створення іншого об'єкта `BinaryTree` і зміни `self.left` або `self.right` кореня так, щоб цей атрибут посилився на нове дерево.

Над структурами типу дерева виконують такі основні дії:

- створення дерева,
- обхід дерева (усіх його вершин),
- додавання вузла у визначене місце в дереві.
- додавання цілого фрагмента дерева.
- видалення вузла з дерева.
- видалення цілого фрагмента дерева.
- трансформації (повороти) фрагментів дерева тощо.

Найпростіший спосіб побудови бінарного дерева полягає у створенні дерева симетричної структури із наперед відомою кількістю вузлів. Усі вузли-нащадки, що створюються, рівномірно розподіляються зліва та справа від кожного вузла-предка. Правило рівномірного розподілу n вузлів можна визначити рекурсивно:

- 1) перший вузол вважати коренем дерева;
- 2) створити ліве піддерево з кількістю вузлів **numberLeft = n/2**;
- 3) створити праве піддерево з кількістю вузлів **numberRight = n - numberLeft - 1**.

При цьому досягається мінімально можлива глибина для заданої кількості вузлів дерева. Наприклад, код C++ для побудови такого дерева на основі елементів масиву m:

// C++

```
Node* Tree :: makeTree(int m[], int from, int n) {
    if (n == 0) return NULL;
    Node* p = new Node(m[from]);
    int nl = n / 2;
    int nr = n - nl - 1;
    p->left= makeTree(m, from + 1, nl);
    p->right= makeTree(m, from + 1 + nl, nr);
    return p;
}
```

Тут n – кількість елементів масиву, які будуть поміщатися у дерево (піддерево), from – індекс елемента масиву, починаючи з якого буде здійснюватися розміщення елементів масиву у дерево (піддерево).

У Python побудова дерев можлива не тільки в стилі ООП, а і як список списків.

Приклад побудови дерева в об'єктно-орієнтованому стилі:

```
class BinaryTree:
    def __init__(self, value):
        self.key = value
        self.left = None
```

```
self.right = None

def insertLeft(self, newNode):
    if self.left == None:
        self.left = BinaryTree(newNode)
    else:
        obj = BinaryTree(newNode)
        obj.left = self.left
        self.left = obj

def insertRight(self, newNode):
    if self.right == None:
        self.right = BinaryTree(newNode)
    else:
        obj = BinaryTree(newNode)
        obj.right = self.right
        self.right = obj

...

```

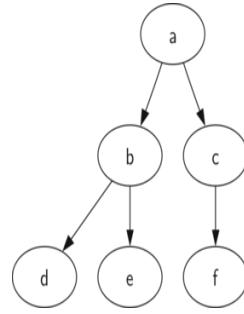
Процедура вставки вузла у дерево передбачає два варіанта. Перший - для вузла, у якого немає лівого (правого) нащадка. У цьому випадку вузол просто вставляється у дерево. Другий варіант стосується вузла, що має лівого (правого) нащадка. Тоді треба вставити новий вузол і спустити наявного нащадка на один рівень нижче. Цей випадок управляється оператором **else**.

У дереві Python, представленаому як список списків, на першій позиції зберігається значення кореневого вузла; другий елемент сам по собі є списком і представляє ліве піддерево; третій елемент є правим піддеревом. Наприклад, дерево і пов'язана з ним спискова реалізація Python:

```

myTree = ['a',      #root
          ['b',        #left subtree
           ['d' [], []],
           ['e' [], []]],
          ['c',        #right subtree
           ['f' [], []],
           []]
        ]

```



При обробці такого дерева можливий доступ до кожного із піддерев з використанням стандартної спискової індексації: корінь дерева - `myTree[0]`, ліве піддерево - `myTree[1]`, праве - `myTree[2]`. Наприклад,

```

print(myTree)

print('left subtree = ', myTree[1])

print('root = ', myTree[0])

print('right subtree = ', myTree[2])

```

Відеокопія результату:

```

Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
[['a', ['b', ['d', [], []], ['e', [], []]], ['c', ['f', [], []], []]]
left subtree = ['b', ['d', [], []], ['e', [], []]]
root = a
right subtree = ['c', ['f', [], []], []]
>>>
Ln: 229 Col: 4

```

Одна з переваг підходу зі списком списків полягає у тому, що структура списку, що представляє піддерево, чітко дотримується визначення дерева - вона рекурсивна сама по собі: у піддерева є корінь і два порожніх списки в якості листя. Інша позитивна якість списку списків полягає в тому, що він легко розширяється до дерева, що має багато піддерев. Тобто в разі, коли дерево не є двійковим, нове піддерево - це всього лише новий підсписок.

Можна спростити використання списків як дерев за рахунок визначення деяких функцій. Наприклад,

```

def BinaryTree(r):
    return [r, [], []]

```

```
def insertLeft (root, newBranch):  
    t = root.pop(1)  
    if len (t)> 1:  
        root.insert (1, [newBranch, t, []])  
    else:  
        root.insert (1, [newBranch, [], []])  
    return root  
  
def insertRight (root, newBranch):  
    t = root.pop(2)  
    if len (t)> 1:  
        root.insert (2, [newBranch, [], t])  
    else:  
        root.insert (2, [newBranch, [], []])  
    return root  
  
def getRootVal(root):  
    return root[0]  
  
def setRootVal(root, newVal):  
    root[0] = newVal  
  
def getLeftChild(root):  
    return root[1]  
  
def getRightChild(root):  
    return root[2]  
  
r = BinaryTree(3)  
insertLeft(r, 4)
```

```

insertLeft(r, 5)

insertRight(r, 6)

insertRight(r, 7)

l = getLeftChild(r)

print(l)

setRootVal(l, 9)

print(r)

insertLeft(l, 11)

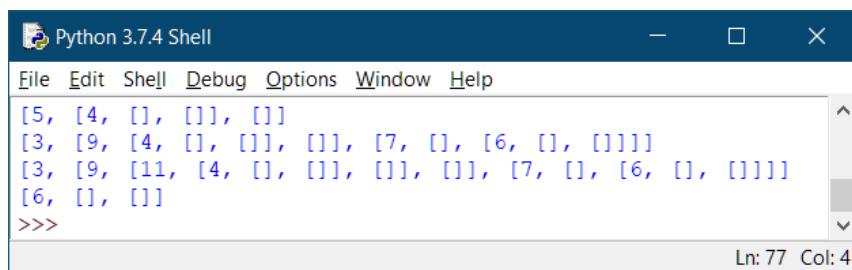
print(r)

print(getRightChild(getRightChild(r)))

```

Тут функція `BinaryTree` просто створює список з кореневого вузла і двох порожніх підсписків як його нащадків. Щоб додати до кореня ліве піддерево, потрібно вставити на другу позицію новий список. Функції `insertLeft` і `insertRight` служать для вставки лівого і правого піддерев.

Відеокопія результата:



The screenshot shows the Python 3.7.4 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following code execution:

```

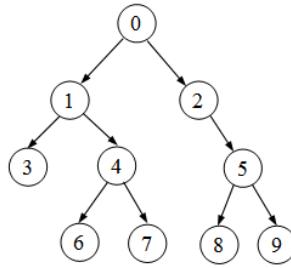
Python 3.7.4 Shell
File Edit Shell Debug Options Window Help
[5, [4, [], []], []]
[3, [9, [4, [], []], []], [7, [], [6, [], []]]]
[3, [9, [11, [4, [], []], []], [7, [], [6, [], []]]]]
[6, [], []]
>>>
Ln: 77 Col: 4

```

7.3 Обхід дерева

Обхід дерева слід проводити за рахунок послідовного виділення в дереві подібних найпростіших піддерев і застосуванням до кожного з них відповідного правила обходу. Виділення починається з коренової вершини.

Як приклад, розглянемо обхід наступного бінарного дерева з числовими компонентами:



Обхід в прямому порядку дає наступний порядок обходу вершин: 0-1-3-4-6-7-2-5-8-9; симетричний обхід: 3-1-6-4-7-0-2-8-5-9; обхід в зворотному порядку: 3-6-7-4-1-8-9-5-2-0.

Результатуюча послідовність вершин і складність обходу істотно залежить від правила обходу.

Враховуючи рекурсивний характер правил обходу, відповідна програмна реалізація може бути виконана за допомогою рекурсивних функцій.

Кожен рекурсивний виклик відповідає за обробку свого поточного піддерева. Після повної обробки поточного піддерева відбувається повернення до піддерева верхнього рівня, а для цього треба запам'ятовувати і надалі відновлювати адресу кореневої вершини цього піддерева. Рекурсивні виклики дозволяють виконати це запам'ятовування і відновлення автоматично, якщо описати адресу кореневої вершини піддерева як формальний параметр рекурсивної функції.

Кожен рекурсивний виклик, насамперед, повинен перевірити передану адресу на `NULL`. Якщо ця адреса дорівнює `NULL`, то чергове оброблюване піддерево є порожнім і його обробка не потрібна, тому просто відбувається повернення з рекурсивного виклику. В іншому випадку, у відповідності до правила обходу, проводиться або обробка вершини, або рекурсивний виклик для обробки лівого чи правого піддерева.

Псевдокод рекурсивної реалізації обходу дерева в прямому напрямку:

```

if (p)
{ <обробка p>
  TLR(p->left);
}
  
```

```
    TLR(p->right);
```

```
}
```

В якості стартової вершини задається адреса кореневої вершини дерева.

Реалізації симетричного і зворотного обходу вершин дерева відрізняються тільки порядком виконання трьох основних інструкцій в тілі умовного оператора:

```
// симетричний прохід           // зворотний прохід  
if (p)  
{   TLR(p->left);  
    <обробка p>  
    TLR(p->right);  
}  
  
if (p)  
{   TLR(p->left);  
    TLR(p->right);  
    <обробка p>  
}  
}
```

Досить легко реалізувати нерекурсивний варіант процедур обходу, якщо врахувати, що рекурсивні виклики та повернення використовують стековий принцип роботи. Наприклад, розглянемо схему реалізації нерекурсивного симетричного обходу. У відповідності із даним правилом, спочатку треба обробити всі ліві нащадки, тобто спуститься вліво максимально глибоко. Кожне просування вниз до лівого нащадка призводить до запам'ятовування в стеку адреси колишньої кореневої вершини. Тим самим для кожної вершини в стеку запам'ятується шлях до цієї вершини від кореня дерева.

Звернення до рекурсивної функції для обробки лівого нащадка треба замінити розміщенням в стек адреси поточної кореневої вершини і переходом до лівого нащадку цієї вершини. Обробка правого нащадка полягає у витяганні із стека адреси деякої вершини і переході до її правого нащадка.

Для нерекурсивного обходу дерева необхідно оголосити допоміжну структуру даних - стек. В інформаційній частині елементів стека повинні зберігатися адреси вузлів цього дерева, тому її треба описати за допомогою

відповідного посилального типу. Псевдокод нерекурсивного симетричного обходу дерева:

```
current = root;                                // починаємо з коренової вершини
дерева

stop = 0;                                         // допоміжна змінна

while (stop == 0)                                 // основний цикл обходу

{ while (current!= NULL)                         // обробка лівих нащадків

{ <занести current в стек>;

    current = current->left;

}

if (<стек порожній>)

    stop = 1;                                     // обхід закінчений

else

{   <витягти з стека адресу і привласнити його current>;

    <обробка вузла current>;

    current = current->right;

}

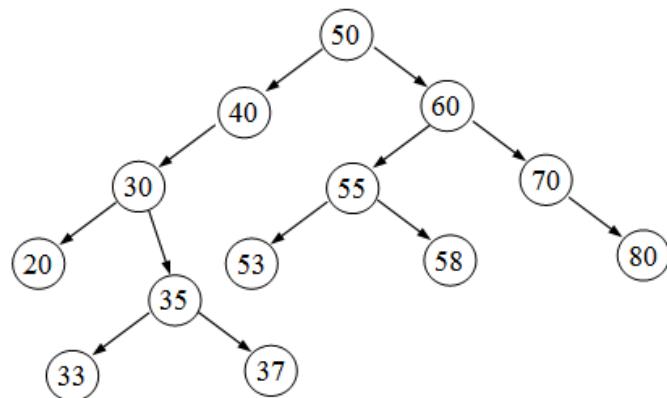
}
```

7.4 Бінарні дерева пошуку

На основі процедур обходу легко можна реалізувати пошук в дереві вершини із заданим інформаційним значенням. Для цього кожна поточна вершина перевіряється на збіг із заданим значенням і в разі успіху відбувається завершення обходу.

Основна сфера використання бінарних дерев – реалізація алгоритмів бінарного пошуку. Такі дерева називають *бінарними деревами пошуку (BST, Binary Search Tree)*.

Двійкове дерево називається *деревом пошуку*, якщо для кожного вузла дерева всі значення в його лівому піддереві менші за значення цього вузла, а всі значення в його правому піддереві більші значення вузла. Наприклад,



Дерева пошуку є однією з найбільш ефективних структур побудови впорядкованих даних. Як відомо, у впорядкованому масиві дуже ефективно реалізується пошук, але дуже важко виконати додавання і видалення елементів. Навпаки, в упорядкованому списку легко реалізується додавання і видалення елементів, але не ефективна реалізація пошуку через необхідність послідовного перегляду всіх елементів, починаючи з першого. Дерева пошуку дозволяють об'єднати переваги масивів і лінійних списків: легко реалізується додавання і видалення елементів, а також ефективно виконується пошук.

Алгоритм пошуку в дереві пошуку дуже простий. Починаючи з кореневої вершини для кожного поточного піддерева треба виконати наступні кроки:

- порівняти ключ вершини із заданим значенням;
- якщо задане значення менше ключа вершини, перейти до лівого нащадку, інакше перейти до правого піддерева.

Пошук припиняється при виконанні однієї з двох умов: або якщо знайдений шуканий елемент, або якщо треба продовжувати пошук в порожньому піддереві, що є ознакою відсутності шуканого елемента.

Слід зазначити, що пошук дуже легко можна реалізувати простим циклом, без використання рекурсії:

```

// root – корінь дерева, value – шукане значення

Node* current = root;           // починаємо пошук з кореня дерева

int stop = 0;

while (current != NULL && stop == 0)

    if (value < current->data) current = current->left;

    else

        if (value > current->data) current = current->right;

        else stop = 1;

```

Трохи складніше реалізується операція додавання нового елемента в дерево пошуку. Перш за все, треба знайти відповідне місце для нового елемента, тому додавання нерозривно пов'язано з процедурою пошуку. Будемо вважати, що в дерево можуть додаватися елементи з однаковими ключами, і для цього з кожною вершиною зв'яжемо лічильник числа появи цього ключа. У процесі пошуку може виникнути одна з двох ситуацій:

- знайдена вершина із заданим значенням ключа і в цьому випадку просто збільшується лічильник;
- пошук треба продовжувати по порожньому посиланню, що говорить про відсутність в дереві шуканої вершини, більше того, тим самим визначається місце в дереві для розміщення нової вершини.

Саме додавання включає наступні кроки:

- 1) виділення пам'яті для нової вершини;
- 2) формування її інформаційної складової;
- 3) формування двох порожніх посилальних полів на майбутніх нащадків;
- 4) формування в батьківській вершині лівого або правого посилального поля - адреси нової вершини.

Тут тільки остання операція викликає деяку складність, оскільки для доступу до посилального поля батьківського вершини треба знати її адресу. Ситуація аналогічна додаванню елемента в лінійний список перед заданим, коли для відстеження елемента-попередника при проході за списком використовувався додатковий покажчик. Цей же прийом можна використовувати і для дерев, але є більш елегантне рішення - рекурсивний

пошук із додаванням нової вершини при необхідності. Кожен рекурсивний виклик відповідає за обробку чергової вершини дерева, починаючи з кореня, а вся послідовність вкладених викликів дозволяє автоматично запам'ятовувати шлях від кореня до будь-якої поточної вершини. Процедура пошуку повинна мати формальний параметр-змінну посилального типу, який відстежує адресу поточної вершини дерева і як тільки ця адреса стає порожньою, створюється нова вершина і її адреса повертається в викликачу функцію, тим самим автоматично формуючи необхідне посилання у батьківської вершини.

У порівнянні із додаванням, видалення реалізується більш складним алгоритмом і передбачає кілька різних ситуацій:

- вузол, що видаляється, не має нащадків (є листком дерева);
- вузол, що видаляється, має одного нащадка;
- вузол, що видаляється, має двох нащадків.

Коли вузол, що видаляється, є листком дерева, слід звільнити ділянку динамічної пам'яті, яку цей вузол займав, та присвоїти значення NULL покажчикові на даний вузол (рис. 17, а). Якщо видаляється вузол, який має одного нащадка, то покажчику на цей вузол слід присвоїти адресу його нащадка і звільнити пам'ять, яку даний вузол займав (рис. 17, б).

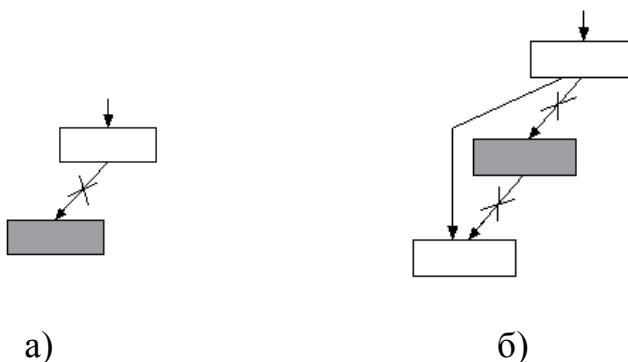


Рис. 17. Схеми видалення вузлів дерева

Якщо вузол, що видаляється, має двох нащадків, на його місце слід переставити інший вузол дерева так, щоб не порушувалася властивість впорядкованості ключів. Тобто, у цьому випадку потрібно знайти підходяще піддерево, яке можна було б вставити на місце вершини, що видаляється. Таке піддерево завжди існує: це або самий правий елемент лівого піддерева, або самий лівий елемент правого піддерева вузла, що видаляється. У першому

випадку для досягнення цієї ланки необхідно перейти в наступну вершину по лівій гілці, а потім переходити в чергові вершини тільки по правій гілці до тих пір, поки чергове поле-вказівник вузла не буде дорівнює `NULL`. У другому випадку необхідно, навпаки, перейти в наступну вершину по правій гілці, а потім переходити в чергові вершини тільки по лівій гілці до тих пір, поки чергове поле-вказівник вузла не стане `NULL`. Зрозуміло, що такі підходячі ланки можуть мати не більше однієї гілки. Нижче (рис. 18) схематично зображенено виключення з дерева вершини з ключем 50:

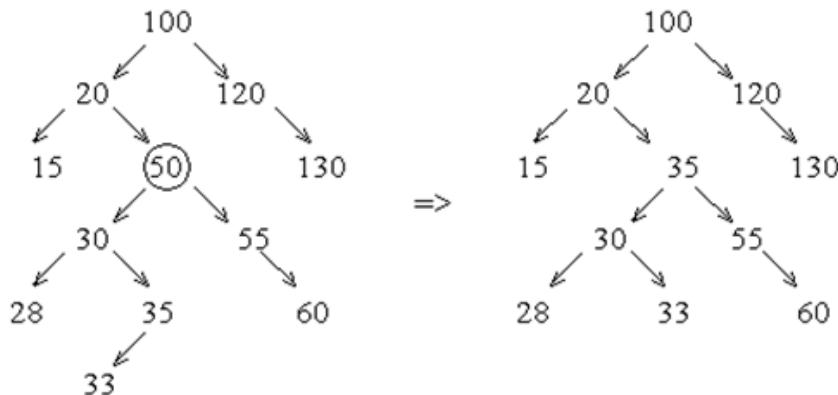


Рис. 18. Приклад видалення вузлів дерева

Приклад метода для видалення вершин BST-дерева:

```

Node* Tree :: DeleteNode(Node* node, int value) {
    if (node == NULL) return node;

    if (value < node->data && node->left) node->left = DeleteNode(node-
>left, value);

    else

        if (value > node->data && node->right) node->right = DeleteNode(node-
>right, value);

    else {

        Node* tmp;

        if (node->right == NULL) tmp = node->left;
        else {
    
```

```

Node* ptr = node->right;

if (ptr->left == NULL) {

    ptr->left = node->left;
    tmp = ptr;
}

else {

    Node* pmin = ptr->left;

    while (pmin->left != NULL) {

        ptr = pmin;
        pmin = ptr->left;
    }

    ptr->left = pmin->right;

    pmin->left = node->left;

    pmin->right = node->right;

    tmp = pmin;

}

delete node;

return tmp;

}

return node;
}

```

7.5 Збалансовані дерева

Ефективне використання дерев на практиці часто вимагає управління ростом дерева для усунення крайніх випадків, коли дерево вироджується в лінійний список і тим самим втрачає всю свою привабливість (з обчислювальної точки зору).

Ідеально збалансовані дерева є ресурсоємними для балансування, тому зазвичай балансування здійснюється для АВЛ-дерев.

Найбільшого розповсюдження АВЛ-дерева набули в тих задачах, де необхідне маніпулювання з ієрархічними даними, ефективний пошук в даних, їхнє структуроване зберігання та модифікація.

Дерево (піддерево), яке потребує балансування, балансирується за допомогою операції обертання вліво чи вправо (переважно при видаленні або додаванні нових елементів).

Збалансоване дерево легко будується, якщо заздалегідь відома кількість вершин n в цьому дереві. У цьому випадку таке дерево можна побудувати за допомогою наступного рекурсивного алгоритму:

- 1) взяти першу по порядку вершину в якості кореневої;
 - 2) знайти кількість вершин у лівих і правих піддеревах: $nl = n/2$; $nr = n-nl-1$;
 - 3) аналогічним чином побудувати ліве піддерево з nl вершинами (поки не отримаємо $nl = 0$)
 - 4) аналогічним чином побудувати праве піддерево з nr вершинами (поки не отримаємо $nr=0$)

Зрозуміло, що реалізація рекурсивного алгоритму виконується у вигляді рекурсивної функції. При цьому між цією процедурою і процедурами обходу є одна принципова відмінність: процедури обходу лише використовують існуючу структуру дерева, не змінюючи її, і тому їх формальні параметри є лише вхідними, тоді як процедура побудови збалансованого дерева повинна створювати вершини і щоразу повернати в викликаючу її функцію адресу чергової створеної вершини. Наприклад,

```
void AddNodes (node **current, int an)
{ node *tmp;
int nl, nr;
if (an == 0) // вершин для размещения нет
    current = NULL; // формируеме порожнє посилання
else
```

```

{ nl = an / 2;           // кількість вершин ліворуч

    nr = an - nl - 1;    // кількість вершин праворуч

    tmp = new node;        // створення кореня піддерева

    AddNodes (& tmp->left, nl) // створення лівого піддерева

    AddNodes (& tmp->right, nr); // створення правого піддерева

    current = tmp;          // повернення адреси створеного кореня

}

}

```

Запуск процесу побудови, зазвичай, виконується з головної програми за допомогою виклику

`AddNodes (&root, n).`

У цьому виклику фактичний параметр `n` обов'язково повинен мати конкретне значення, наприклад - задану користувачем кількість вершин в дереві, що будується. Однак, перший фактичний параметр `root`, будучи вихідним, отримає своє значення лише після відпрацювання всіх рекурсивних викликів, при поверненні в головну програму.

7.6 Купи (піраміди)

Купа, або піраміда (heap) в інформатиці — спеціалізована деревовидна структура даних, в якій існують певні властивості впорядкованості: якщо `B` — вузол нащадок `A` — тоді $\text{ключ}(A) \geq \text{ключ}(B)$.

З цього випливає, що елемент з найбільшим ключем завжди є кореневим вузлом. Не існує ніяких обмежень щодо максимальної кількості елементів-нащадків які повинна мати кожна ланка, однак, на практиці, зазвичай, кожен елемент має не більше двох нащадків.

Купа є однією із найефективніших реалізацій абстрактного типу даних, який має називу черга з пріоритетом.

Купи відіграють критичну роль у низці ефективних алгоритмів роботи з графами, як то в алгоритмі Дейкстри та в алгоритмі сортування піраміdalne сортування.

Найуживанішим класом куп є бінарні купи.

Базові операції з купою такі:

- підтримка основної властивості купи;
- побудова купи з невпорядкованого масиву;
- сортування купи;
- видалення найменшого елемента;
- отримання найбільшого елемента;
- додавання елемента.

Купи часто використовуються для моделювання черг з пріоритетами.

7.6.1 Двійкова купа

Двійкова купа, піраміда, або сортуоче дерево - таке бінарне дерево, для якого виконуються три умови:

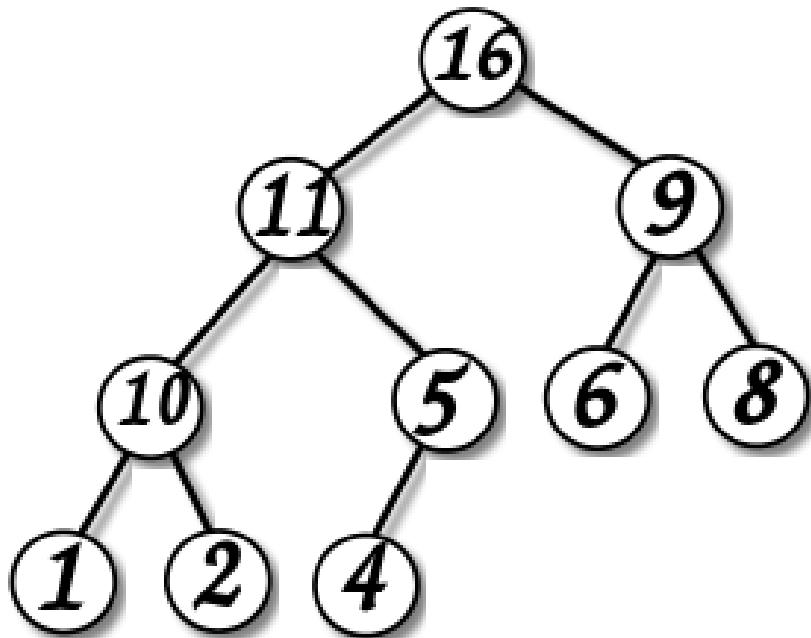
- Значення в будь-якій вершині не менш, ніж значення її нащадків.
- Глибина усіх листків (відстань до кореня) відрізняється не більше ніж на 1 шар.
- Останній шар заповнюється зліва направо без «дірок».

Існують також купи, де значення в будь-якій вершині, навпаки, не більше, ніж значення її нащадків. Такі купи називаються min-heap, а купи, описані вище - max-heap. Надалі розглядаються тільки max-heap. Всі дії з min-heap здійснюються аналогічно.

Зручна структура даних для сортування дерева - масив A, у якого перший елемент, A [1] - елемент в корені, а нащадками елемента A [i] є A [2i] і A [2i + 1] (при нумерації елементів з першого) . При нумерації елементів з нульового,

кореневий елемент - A [0], а нащадки елемента A [i] - A [2i + 1] і A [2i + 2]. При такому способі зберігання умови 2 і 3 виконані автоматично.

Висота купи визначається як висота двійкового дерева. Тобто вона дорівнює кількості ребер в самому довгому простому шляху, що з'єднує корінь купи з одним з її листків. Висота купи є $\Theta(\log \lg N)$, де N - кількість вузлів дерева.



16	11	9	10	5	6	8	1	2	4
----	----	---	----	---	---	---	---	---	---

Основні операції:

- Відновлення властивостей купи
- Побудова купи
- Зміна значення елемента
- Додавання елемента
- Отримання максимального елемента

Відновлення властивостей купи

Якщо в купі змінюється один з елементів, то вона може перестати задовольняти властивості впорядкованості. Для відновлення цієї властивості служить процедура Heapify.

Вона відновлює властивість купи в дереві, у якої ліве і праве піддерева задовольняють їй.

Ця процедура приймає на вхід масив елементів A і індекс i. Вона відновлює властивість впорядкованості у всьому піддереві, коренем якого є елемент A [i].

Якщо i-й елемент більше, ніж його сини, все піддерево вже є купою, і робити нічого не треба. В іншому випадку міняємо місцями i-й елемент з найбільшим з його синів, після чого виконуємо Heapify для цього сина.

Процедура виконується за час $O(\log n)$.

Heapify(A, i)

left $\leftarrow 2i$

right $\leftarrow 2i+1$

heap_size – кількість елементів

largest $\leftarrow i$

if $left \leq A.heap_size$ **и** $A[left] > A[largest]$

then $largest \leftarrow left$

if $right \leq A.heap_size$ **и** $A[right] > A[largest]$

then $largest \leftarrow right$

if $largest \neq i$

then swap $A[i] \leftrightarrow A[largest]$

Heapify(A, largest)

Побудова купи

Ця процедура призначена для створення купи з невпорядкованого масиву вхідних даних.

Зауважимо, що якщо виконати Heapify для всіх елементів масиву A, починаючи з останнього і закінчуючи першим, він стане купою.

Крім того, Heapify (A, i) не робить нічого, якщо $i > N / 2$ (при нумерації з першого елемента), де N - кількість елементів масиву. Справді, у таких елементів немає нащадків, отже, відповідні піддерева вже є купами, так як містять всього один елемент.

Таким чином, досить викликати Heapify для всіх елементів масиву A, починаючи (при нумерації з першого елемента) з $[N / 2]$ -го і кінчаючи першим.

Build_Heap(A)

```
A.heap_size ← A.length  
for i ← [A.length/2] downto 1  
    do Heapify(A, i)
```

Зміна значення елемента

Процедура Heap_Increase_Key замінює елемент купи на новий із значенням, не менше значення старого елемента.

Зазвичай ця процедура використовується для додавання довільного елемента в купу. Часова складність $O(\log \frac{n}{2} n)$.

Якщо елемент менше свого батька, умова 1 дотримано для всього дерева, і більше нічого робити не потрібно. Якщо він більше, ми міняємо місцями його з батьком. Якщо після цього батько більше діда, ми міняємо місцями батька з дідом і так далі. Іншими словами, найбільший елемент спливає на вершину.

Heap_Increase_Key(A, i, key)

```
if key < A[i]  
    then error «Ключ менше попереднього»  
    A[i] ← key  
    while i > 1 и A[|i/2|] < A[i]
```

do Обменяйтъ A[i] ↔ A[i/2]

i ← [i/2]

У разі, коли необхідно зменшити значення елемента, можна викликати Heapify.

Додавання елемента

Виконує додавання елемента в купу за час $O(\log \frac{n}{\delta} n)$.

Додавання довільного елемента в кінець купи, і відновлення властивості впорядкованості за допомогою Heap_Increase_Key.

Heap_Insert(A, key)

A.heap_size ← A.heap_size+1

A[A.heap_size] ← -∞

Heap_Increase_Key(A, A.heap_size, key)

Отримання максимального елемента

Отримує максимальний елемента з купи за час $O(\log \frac{n}{\delta} n)$.

Виконується в 4 етапи:

- значення кореневого елемента (він є максимальним) зберігається для подальшого повернення
- останній елемент копіюється в корінь, після чого видаляється з купи
- викликається Heapify для кореня
- збережений елемент повертається

Heap_Extract_Max(A)

if A.heap_size[A] < 1

then error "Купа пуста"

max ← A[1]

A[1] ← A[A.heap_size]

A.heap_size ← A.heap_size-1

Heapify(A, 1)

return max

8 ДОДАТОК З – АЛГОРИТМИ ПОШУКУ У СТРУКТУРАХ ДАНИХ

Пошук – це обробка деякої множини даних з метою виявлення підмножини даних, що відповідає критеріям пошуку.

Всі алгоритми пошуку діляться на:

- пошук в неупорядкованій множині даних;
- пошук в упорядкованій множині даних.

Впорядкованість - це наявність відсортованого ключового поля.

8.1 Класичні алгоритми пошуку

8.1.1 Пошук в неупорядкованих множинах даних

8.1.1.1 Лінійний, послідовний пошук

Лінійний, послідовний пошук – алгоритм знаходження заданого значення довільної функції на деякому відрізку. Даний алгоритм є найпростішим алгоритмом пошуку і, на відміну, наприклад, від бінарного пошуку, не накладає ніяких обмежень на функцію і має найпростішу реалізацію. Пошук значення функції здійснюється простим порівнянням почергового розглянутого значення (як правило, пошук відбувається зліва направо, тобто від менших значень аргументу до найбільших) і, якщо значення збігаються (з тією або іншою точністю), то пошук вважається завершеним.

Якщо відрізок має довжину N , то знайти рішення з точністю до ϵ можна за час N / ϵ . (тобто асимптотична складність алгоритму $O(n)$)

8.1.1.2 Метод транспозиції

Поліпшенням розглянутого методу є метод транспозиції: кожен запит до запису супроводжується зміною місць цієї і попереднього запису; в результаті найбільш часто використовувані записи поступово переміщаються в початок таблиці; і при наступному зверненні до них, ці записи знаходяться майже відразу.

8.1.1.3 Метод переміщення на початок

В цьому методі кожен запис до запису супроводжується його переміщенням в початок таблиці. В результаті на початку таблиці виявляється запис, що використовується в останній раз.

8.1.2 Пошук в упорядкованих множинах даних

8.1.2.1 Індексний-послідовний пошук

Для індексного-послідовного пошуку окрім відсортованої таблиці вводиться допоміжна таблиця, яку називають індексною. Кожен елемент індексної таблиці складається з ключа і покажчика на запис в основній таблиці, що відповідає цьому ключу. Елементи в індексній таблиці, як елементи в основній таблиці, повинні бути відсортовані за цим ключем.

Якщо індекс має розмір, що становить $1/8$ від розміру основної таблиці, то кожен восьмий запис основної таблиці буде представлений в індексній таблиці.

Якщо розмір основної таблиці n , то розмір індексної таблиці $=n/8$.

Перевага алгоритму індексно-послідовного пошуку полягає в тому, що скорочується час пошуку, так як послідовний пошук спочатку ведеться в індексній таблиці, що має менший розмір, ніж основна таблиця. Коли знайдений правильний індекс, другий послідовний пошук виконується для невеликої частини записів основної таблиці.



8.1.2.2 Двійковий (бінарний) пошук

Двійковий (бінарний) пошук – класичний алгоритм пошуку елемента в відсортованому масиві (векторі), який використовує дроблення масиву на половини.

Пошук елемента в відсортованому масиві.

- Визначення значення елемента в середині структури даних. Отримане значення порівнюється з ключем.
- Якщо ключ менше значення середини, то пошук здійснюється в першій половині елементів, інакше – в другій.
- Пошук зводиться до того, що знову визначається значення серединного елемента в обраній половині і порівнюється з ключем.
- Процес триває до тих пір, поки не буде знайдений елемент зі значенням ключа або не стане порожнім інтервал для пошуку.

Незважаючи на те, що код досить простий, в ньому є кілька пасток.

Що буде якщо `first` та `last` окремо вміщаються в свій тип, а `first+last` – ні? Якщо теоретично можливі масиви настільки великого розміру, доводиться йти на хитрощі:

- Використовувати код `first + (last - first) / 2`, який точно не приведе до переповнення (те є інше – невід'ємні цілі числа).
- Якщо `first` і `last` – покажчики або ітератори, такий код єдино правильний. Перетворення в `uintptr_t` і подальший розрахунок в цьому типі порушує абстракцію, і немає гарантії, що результат залишиться коректним покажчиком. Зрозуміло, щоб зберігалася складність алгоритму, потрібні швидкі операції «ітератор + число → ітератор», «ітератор-ітератор → число».
- Якщо `first` і `last` – типи зі знаком, провести розрахунок в беззнаковому типі: `((unsigned)first + (unsigned)last) / 2`.
- Написати розрахунок на асемблері, з використанням прапора перенесення. Щось на зразок `add eax, b; rcr eax, 1`. А ось довгі типи використовувати недоцільно, `first + (last - first) / 2` швидше.

У двійковому пошуку часті помилки на одиницю. Тому важливо протестувати такі випадки: порожній масив ($n=0$), шукаємо відсутнє значення (занадто велика, занадто маленьке і десь в середині), шукаємо перший і останній елемент. Чи не виходить алгоритм за межі масиву? Чи не зациклюється?

Іноді потрібно, щоб, якщо шуканий елемент в послідовності існує в кількох примірниках, знаходило не будь-який, а обов'язково перший (як варіант: останній; або взагалі не шуканий елемент, а наступний за ним елемент). Код на С в такій ситуації знаходить перший з рівних, більш простий код на C++ – будь-який.

Вчений Йон Бентлі стверджує, що 90% студентів, розробляючи двійковий пошук, забувають врахувати якусь з цих вимог. І навіть в код, написаний самим Йоном, вкралася помилка: код не стійкий до переповнень.

Приклад.

Маємо наступну послідовність чисел, необхідно знайти номер п/п з ключем 19.

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

$$l = 1, u = 12$$

$$i = (l + u)/2 = 6 \text{ (округляємо до меншого)}$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Ключ менше шуканого, тому $l = i+1 = 7, u = 12$

$$i = (l + u)/2 = 9 \text{ (округляємо до меншого)}$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Ключ більше шуканого, тому $u = i - 1 = 8, l = 7$

$$i = (l + u)/2 = 7 \text{ (округляємо до меншого)}$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Знайшли $i = 7$ (19 ключ)

8.1.2.3 Однорідний бінарний пошук

Модифікацією бінарного пошуку є однорідний бінарний пошук, в якому використовуються тільки два покажчика - поточний стан i та величину його зміни δ . Після кожного порівняння визначаються нові i та δ

$$i = i \pm ([\delta/2] + 1),$$

$$\delta = ([\delta/2]).$$

При парному числі елементів N алгоритм може розглядати фіктивний ключ K_{n+1} , встановлений рівним $+\infty$ (або будь-якій величині, більше заданого ключа K). Необхідно передбачити подібний випадок. Спочатку поточний стан визначається як $[N / 2] + 1$, а зсув як $[N / 2]$. Пошук закінчується невдало, якщо зсув δ стає рівним 0.

Приклад.

Маємо наступну послідовність, необхідно знайти номер п/п з ключем 27.

№ п/п	1	2	3	4	5	6	7	8	9	10	11
Послідовність	2	5	8	9	12	16	19	20	23	25	27

$$i = [N/2] + 1 = [11/2] + 1 = 6$$

$$\delta = [N/2] = 5$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11
Послідовність	2	5	8	9	12	16	19	20	23	25	27

Ключ менше шуканого, тому:

$$i = i + ([\delta/2] + 1) = 6 + ([5/2] + 1) = 9$$

$$\delta = [\delta/2] = 2$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11
Послідовність	2	5	8	9	12	16	19	20	23	25	27

Ключ менше шуканого, тому:

$$i = i + ([\delta/2] + 1) = 9 + ([2/2] + 1) = 11$$

$$\delta = [\delta/2] = 1$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11
Послідовність	2	5	8	9	12	16	19	20	23	25	27

Знайшли $i = 11$ (27 ключ)

Розглянемо випадок парного числа елементів.

Приклад.

Маємо наступну послідовність, необхідно знайти номер п/п з ключем 35.

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

$$i = \lceil N/2 \rceil + 1 = \lceil 12/2 \rceil + 1 = 7$$

$$\delta = \lceil N/2 \rceil = 6$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Ключ менше шуканого, тому:

$$i = i + (\lceil \delta/2 \rceil + 1) = 7 + (\lceil 6/2 \rceil + 1) = 11$$

$$\delta = \lceil \delta/2 \rceil = 3$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Ключ менше шуканого, тому:

$i = i + (\lceil \delta/2 \rceil + 1) = 11 + (\lceil 3/2 \rceil + 1) = 13$ потрапили в фіктивний простір зі значенням ключа $+\infty$

$$\delta = \lceil \delta/2 \rceil = 1$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12	13
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35	$+\infty$

Ключ більше шуканого, тому:

$$i = i - (\lceil \delta/2 \rceil + 1) = 13 - (\lceil 1/2 \rceil + 1) = 12$$

$\delta = \lceil \delta/2 \rceil = 0$, кінець алгоритму.

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12	13
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35	$+\infty$

Знайшли $i = 12$ (35 ключ), якби елемент був відсутній, то алгоритм би завершився невдало.

8.1.2.4 Метод Шарра

Спочатку порівнюється K і K_i , де $i=2^k$, $k = \lceil \log_2 N \rceil$. Якщо $K < K_i$, використовується однорідний бінарний пошук з послідовністю $\delta = 2^{k-1}, 2^{k-2}, \dots, 1, 0$. При $K > K_i$, і $N > 2^k$, встановлюємо $i = i' = N + 1 - 2^I$, де, $I = \lceil \log_2(N - 2^k + 1) \rceil$ і роблячи вигляд, що першим було порівняння $K > K_i$, використовуємо однорідний пошук з $\delta = 2^{I-1}, 2^{I-2}$.

Приклад.

Маємо наступну послідовність, необхідно знайти номер п/п з ключем 19.

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

$$k = \lceil \log_2 N \rceil = \lceil \log_2 12 \rceil = 3,585$$

$$k = 3$$

$$i = i^k = 8$$

$K_i = 20$ йдемо по першому варіанту

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

$$i = i^k = 8$$

$$\delta = 2^{k-1} = 4$$

Ключ більше шуканого, тому:

$$i = i - (\lceil \delta / 2 \rceil + 1) = 8 - (\lceil 4 / 2 \rceil + 1) = 5$$

$$\delta = 2^{k-2} = 2$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Ключ менше, шуканого тому:

$$i = i + ([\delta/2] + I) = 5 + ([2/2]+I) = 7$$

$$\delta = 2^{k-3} = 1$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Знайшли $i = 7$ (19 ключ)

Приклад.

Пример. Маємо наступну послідовність, необхідно знайти номер п/п з ключем 35.

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

$$k = [\log_2 N] = [\log_2 12] = 3,585$$

$$k = 3$$

$$i = i^k = 8$$

$K_i=20$ йдемо по другому варіанту

Оскільки $N > 2^k$

$$l = [\log_2(N - 2^k + 1)] = [\log_2(12 - 2^3 + 1)] = 2$$

$$i = i^k = N + 1 - 2^l = 9$$

$$\delta = 2^{l-1} = 2$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Ключ менше, шуканого тому:

$$i = i + ([\delta/2] + I) = 9 + ([2/2]+I) = 11$$

$$\delta = 2^{l-2} = 1$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Ключ менше, шуканого тому:

$$i = i + (\lceil \delta/2 \rceil + 1) = 11 + (\lceil 1/2 \rceil + 1) = 12$$

$$\delta = 0$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Знайшли $i = 12$ (35 ключ)

8.1.2.5 Пошук Фібоначчі

Працює для відсортованої послідовності елементів. Запропонований метод включає в себе тільки додавання і віднімання, на відміну від бінарного, в якому вимагалося ділення на 2. Числа Фібоначчі використовуються в якості індексів елементів, що визначають межі, в яких може розташовуватися необхідний ключ.

Для спрощення реалізації методу будемо вважати, що $N + 1$ (N - число елементів) число Фібоначчі F_{k+1} .

Нижче наводиться алгоритм пошуку Фібоначчі:

1. Початкова установка.

Встановити $I = F_k$, $p = F_{k-1}$, $q = F_{k-2}$. Тобто p і q -послідовні числа Фібоначчі.

2. Порівняння.

Якщо $K < K_i$, то перейти на крок 3; якщо $K > K_i$ то перейти на крок 4, якщо $K = K_i$ алгоритм закінчується успішно.

3. Зменшення i

Якщо $q=0$, алгоритм закінчується неуспішно. Якщо $q!=0$, то встановити $i=i-q$, замінити (p,q) на $(q,p-q)$. повернутися на крок 2.

4. Збільшення i

Якщо $p = 1$, алгоритм закінчується невдало. Якщо $p \neq 0$, то встановити $i=i+q$, $p=p-q$, $q=q-p$, повернутися на крок 2.

Приклад.

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Число послідовності $N=12$, $N+1=13=F_{k+1}=F_7$, $k=6$. Запишемо перших 6 чисел Фібоначчі:

№ п/п	1	2	3	4	5	6
Число Фібоначчі	1	1	2	3	5	8

Будемо шукати запис з ключем $K=16$

$i=F_6=8$, $p=F_5=5$, $q=F_4=3$. $K_8=20$, $K=16$, тобто зменшуємо i . $i=8-3=5$, $p=3$, $q=2$.

$K_5=12$, збільшуємо i : $i=5+2=7$, $p=1$, $q=1$.

$K_7=19$, зменшуємо i : $i=7-1=6$, $p=1$, $q=0$.

$K_6=16$, пошук завершився вдало.

8.1.2.6 Інтерполяційний пошук

Також застосовується для відсортованої послідовності елементів. Є найбільш природним для людини. Якщо відомо, що необхідний ключ розташовується між ключами K_l і K_u , то наступна спроба проводиться на відстані

$$\frac{(u - l)(K - K_l)}{K_u - K_l}$$

від l , при цьому передбачається, що ключі є числами, що зростають приблизно як арифметична прогресія. В якості вихідних l та u беруться початковий і кінцевий елементи масиву.

Приклад.

Маємо наступну послідовність, необхідно знайти номер п/п з ключем 16.

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

$$i = l + \frac{(u-l)(K - K_l)}{K_u - K_l} = 0 + \frac{(12-1) \cdot (16-2)}{(35-2)} = 4,66 = 4$$

Ключ менше шуканого, тому:

$$l = i + 1 = 5$$

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

$$i = l + \frac{(u-l)(K - K_l)}{K_u - K_l} = 5 + \frac{(12-5) \cdot (16-12)}{(35-12)} = 6,22 = 6$$

Знайшли $i = 6$ (16 ключ)

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

Якщо знайдений на цій ітерації індекс більше ніж шуканий то:

$$u = i - 1;$$

8.2 Хешування

Хешування (англ. **hashing**) – перетворення вхідного масиву даних довільної довжини у вихідний бітовий рядок фіксованої довжини. Такі перетворення також називаються **хеш-функціями**, або **функціями згортання**, а їхні результати називають **хешем**, **хеш-кодом**, **хеш-сумою**, або дайджестом повідомлення.

8.2.1 Таблиці з прямою адресацією

Хеш-таблиця представляє собою узагальнення звичайного масиву. Можливість прямої індексації елементів звичайного масиву дозволяє доступ до довільної позиції в масиві за час $O(1)$.

Загалом пряма індексація представляє собою технологію, яка добре працює для невеликих множин ключів. Припустимо, що застосуванню потрібна динамічна множина, кожний елемент якої має клуз з множини $U = \{0, 1, \dots, m - 1\}$, де m не дуже велике. Крім того, припускається, що жодні два елементи не мають одинакових ключів.

Для представлення динамічної множини використовується масив, або таблиця з прямою адресацією, який позначається як $T[0 \dots m - 1]$, кожна позиція, або комірка, якого відповідає клочу з простору ключів U . На рисунку, 8.1 представлений цей підхід. Комірка k вказує на елемент множини з ключем k . Якщо множина не містить елементу з таким ключем, то $T[k] = \text{NULL}$. На рисунку кожний клоч простору $U = \{0, 1, \dots, 9\}$ відповідає індексу таблиці. Множина реальних ключів $K = \{2, 3, 5, 8\}$ визначає комірки таблиці, які містять покажчики на елементи. Решта комірок містять значення NIL.

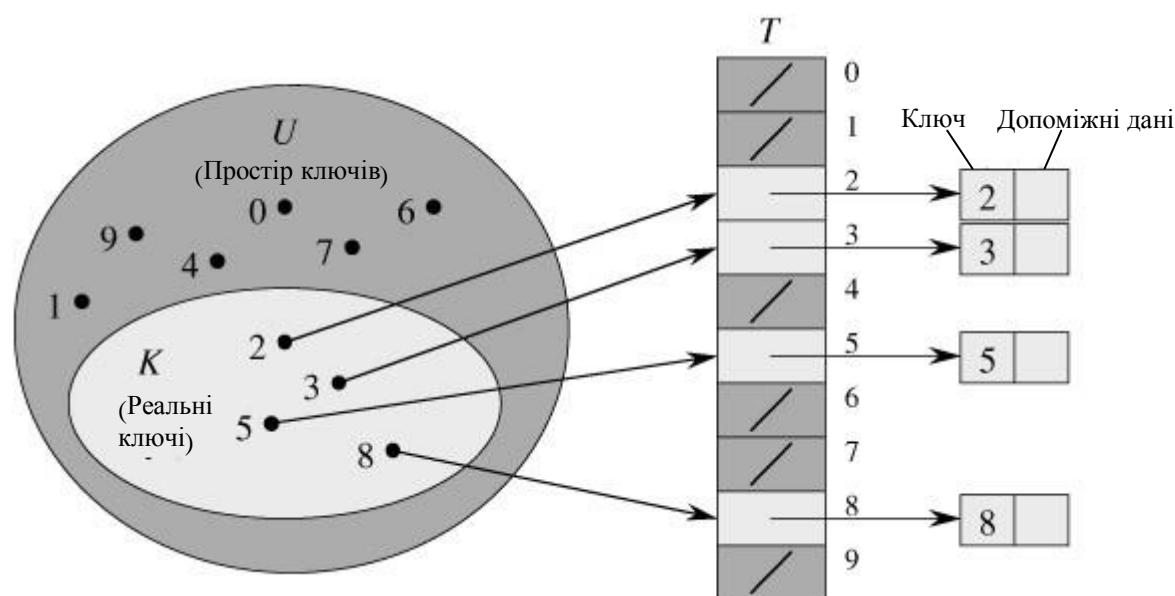


Рис. 8.1. Динамічна множина із використанням таблиці з прямою адресацією.

Реалізація основних операцій в масиві тривіальна.

```
DirectAddressSearch(T, k)
1 return T[k]
```

```
DirectAddressInsert(T, x)
1 T[key[x]] = x
```

```
DirectAddressDelete(T, x)
1 T[key[x]] = NIL
```

Лістинг 3.1 Процедури, які реалізують операції роботи з масивами.

Кожна з наведених операцій дуже швидка: час їх роботи дорівнює $O(1)$.

В деяких застосуваннях елементи динамічної множини можуть зберігатись безпосередньо в таблиці з прямою адресацією. Тобто замість зберігання ключів та допоміжних даних елементів в об'єктах, які є зовнішніми по відношенню до таблиці з прямою адресацією, а в таблиці – покажчиків на ці об'єкти, ці об'єкти можна зберігати безпосередньо в комірках таблиці (що призводить до економії пам'яті).

8.2.2 Хеш-таблиці

Недолік прямої адресації очевидний: якщо простір ключів U великий, то зберігання таблиці T розміром $|U|$ непрактично, а для деяких задач навіть неможливо. Крім того, множина K реальних ключів може бути малою по відношенню до U , а в цьому випадку пам'ять, яка виділена під T , здебільшого витрачається дарма.

Коли множина K ключів, які зберігаються у словнику, значно менша простору U , хеш-таблиця вимагає значно менше місця, чим таблиця з прямою адресацією. Точніше кажучи, вимоги до пам'яті можуть бути знижені до $\Theta(|K|)$, при цьому час пошуку елементу в хеш-таблиці лишається рівним $O(1)$. Потрібно тільки відзначити, що це межа середнього часу пошуку, в той час як у випадку таблиці з прямою адресацією ця межа справедлива для найгіршого випадку.

У випадку прямої адресації елемент з ключем k зберігається у комірці k . При хешуванні цей елемент зберігається в комірці $h(k)$, тобто тут використовується **хешфункція h** для обчислення комірки для даного ключа k . Функція h відображає простір ключів U на комірки **хеш-таблиці** $T[0 \dots m - 1]$:

$$h: U \rightarrow \{0, 1, \dots, m - 1\}.$$

Ми говоримо, що елементи з ключем k хешуються в комірку $h(k)$; величина $h(k)$ називається хеш-значенням ключа k . На рис. 8.2 представлена основна ідея хешування. Мета хеш-функції полягає в тому, щоб зменшити робочий діапазон індексів масиву, а замість $|U|$ значень ми можемо обійтись лише m значеннями. Відповідно знижуються вимоги до об'єму пам'яті.

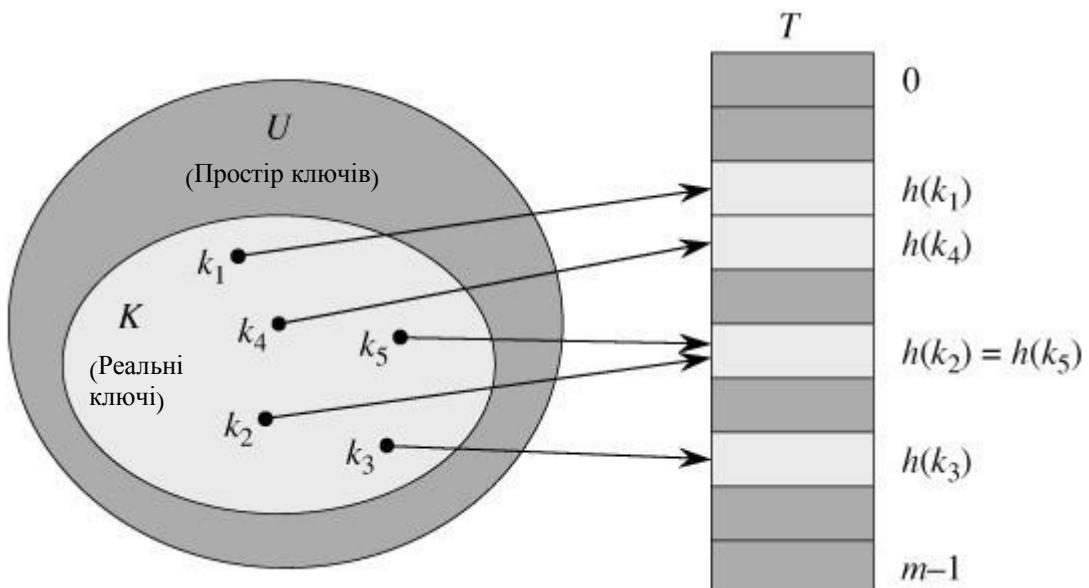


Рис. 8.2. Використання хеш-функції для відображення ключів у комірки хеш-таблиці.

Проте тут є одна проблема: два ключа можуть мати одне й те саме хеш-значення. Так ситуація називається **колізією**. На щастя є ефективні технології для розв'язання конфліктів, які виникають під час колізій.

Звісно, ідеальним рішенням було би повне уникнення колізій. Цього можна досягнути шляхом підбора відповідної хеш-функції. Одна з ідей полягає в тому, щоб зробити функцію h випадковою. Зрозуміло, що хеш-функція мусить бути детермінованою і для одного й того самого значення k завжди давати одне й те саме значення $h(k)$. Однак оскільки $|U| > m$, повинні існувати як мінімум два ключа, які мають однакове хеш-значення. Таким чином,

повністю уникнути колізій принципово неможливо, і добра хеш-функція може тільки мінімізувати кількість колізій.

8.2.3 Уникнення колізій за допомогою ланцюгів

За допомогою методу ланцюгів всі елементи, які хешуються в одну й ту саму комірку, об'єднуються у зв'язаний список, як це показано на рис. 8.3. Комірка j містить покажчик на заголовок списку всіх елементів, хеш-значення ключа яких дорівнює j ; якщо таких елементів немає, комірка містить значення NIL. На рис. 8.3 наведене розв'язання колізій, які виникають через те, що $h(k_1) = h(k_4)$, $h(k_5) = h(k_2) = h(k_7)$ та $h(k_8) = h(k_6)$.

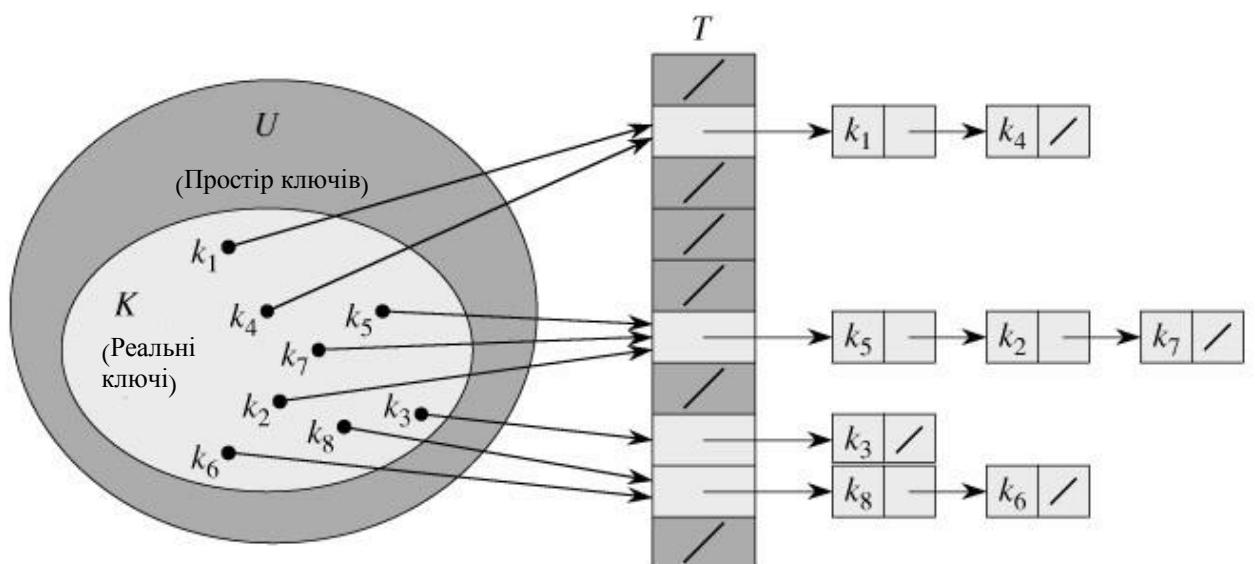


Рис. 8.3. Розв'язання колізій за допомогою ланцюгів.

Словарні операції в хеш-таблиці із використанням ланцюгів для розв'язання колізій реалізуються дуже просто:

`ChainedHashInsert(T, x)`

1 Вставити x в заголовок списку $T[h(key[x])]$

`ChainedHashSearch(T, k)`

1 Пошук елементу з ключем k в списку $T[h(k)]$

`ChainedHashDelete(T, x)` 1

Видалення x зі списку $T[h(key[x])]$

Лістинг 8.2. Процедури, які реалізують операції роботи з хеш-таблицями.

Час, необхідний для вставки в найгіршому випадку, дорівнює $O(1)$. Процедура вставки виконується дуже швидко, адже передбачається, що елемент для вставки відсутній у таблиці. При необхідності це припущення може бути перевірено виконанням пошуку перед вставкою. Час роботи пошуку в найгіршому випадку пропорційний довжині списку. Видалення елементу може бути виконане за час $O(1)$ при використанні двозв'язаних списків.

Наскільки висока ефективність хешування з ланцюгами? Зокрема, скільки часу потрібно для пошуку елементу з даним ключем?

Нехай ми маємо хеш-таблицю T з m комірками, в яких зберігаються n елементів.

Визначимо **коєфіцієнт заповнення** таблиці T як $\alpha = n/m$, тобто як середню кількість елементів, які зберігаються в одному ланцюгу. Подальший аналіз буде ґрунтуватись на значенні величини α , яка може бути менше, дорівнювати або більше одиниці.

В найгіршому випадку хешування з ланцюгами поводить себе досить неприємно: всі n ключів хешуються в одну й ту саму комірку, створюючи при цьому список довжиною n . Таким чином, час пошуку в найгіршому випадку дорівнює $\Theta(n)$ плюс час обрахунку хеш-функції, що нічим не краще використання зв'язаного списку для збереження всіх n елементів. Зрозуміло, що використання хеш-таблиць в найгіршому випадку немає сенсу.

Середня продуктивність хешування залежить від того, наскільки добре хеш-функція h розподіляє множину ключів для зберігання по m коміркам в середньому. Ми розглянемо це питання детальніше згодом, а зараз припустимо, що всі елементи хешуються по комірках рівномірно та незалежно, і назовемо це припущення «**простим рівномірним хешуванням**».

Позначимо довжини списків $T[j]$ для $j = 0, 1, \dots, m - 1$ як n_j , так що

$$n = n_0 + n_1 + \dots + n_{m-1},$$

а середнє значення n_j дорівнює $E[n_j] = \alpha = n/m$.

Будемо вважати, що хеш-значення $h(k)$ може бути обраховане за час $O(1)$, так що час, необхідний для пошуку елементу з ключем k , лінійно залежить від довжини $n_{h(k)}$ списку $T[h(k)]$. Розглянемо математичне сподівання кількості елементів, яке повинно бути перевірене алгоритмом пошуку (тобто кількість елементів в списку $T[h(k)]$, які перевіряються на рівність їх ключів величині k). Ми повинні розглянути два випадки: поперше, коли пошук невдалий і в таблиці немає елементів з ключем k , і, по-друге, коли пошук закінчується вдало і в таблиці виявлений елемент з ключем k .

Теорема 8.1. В хеш-таблиці з розв'язком колізій методом ланцюгів математичне сподівання часу невдалого пошуку у припущені простого рівномірного хешування дорівнює $\Theta(1 + \alpha)$.

Доведення. У припущені простого рівномірного хешування довільний ключ k , який ще не знаходиться у таблиці, може бути розміщений з однаковою ймовірністю в будь-яку з m комірок. Математичне сподівання часу невдалого пошуку ключа k дорівнює часу пошуку до кінця списку $T[h(k)]$, математичне сподівання довжини якого $E[n_j] = \alpha$. Таким чином, при невдалому пошуку математичне сподівання кількості елементів, які перевіряються, дорівнює α , а загальний час, необхідний для пошуку, включаючи час обрахунку хеш-функції $h(k)$, дорівнює $\Theta(1 + \alpha)$. ■

Вдалий пошук дещо відрізняється від невдалого, оскільки ймовірність пошуку в списку відмінна для різних списків та пропорційна кількості елементів, які містяться в ньому. Тим не менш, і в цьому випадку математичне сподівання часу пошуку лишається рівним $\Theta(1 + \alpha)$.

Теорема 8.2. В хеш-таблиці з розв'язком колізій методом ланцюгів математичне сподівання часу вдалого пошуку у припущені простого рівномірного хешування дорівнює $\Theta(1 + \alpha)$.

Доведення. Будемо вважати, що шуканий елемент з однаковою ймовірністю може бути довільним елементом, який зберігається в таблиці. Кількість елементів, які перевіряються в процесі вдалого пошуку елементу x , на 1 більше, ніж кількість елементів, які знаходяться у списку перед x .

Елементи, які знаходяться в списку до x , були вставлені в список після того, як елемент x був збережений в таблиці, адже нові елементи додаються у початок списку. Для того щоб знайти математичне сподівання кількості елементів, які перевіряються, візьмемо середнє значення по всім елементам таблиці, яке дорівнює 1 плюс математичне сподівання кількості елементів, доданих у список після шуканого.

Нехай x_i означає i -й елемент, який вставлений у таблицю ($i = 1, 2, \dots, n$), а $k_i = \text{key}[x_i]$. Визначимо для ключів k_i та k_j індикаторну випадкову величину $X_{ij} = I\{h(k_i) = h(k_j)\}$. За припущенням простого рівномірного хешування, $P(h(k_i) = h(k_j)) = 1/m$ і, відповідно, $E[X_{ij}] = 1/m$. Таким чином, математичне сподівання кількості елементів, які перевіряються, у випадку вдалого пошуку дорівнює:

$$\begin{aligned} E \left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=1+i}^n X_{ij} \right) \right] &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=1+i}^n E[X_{ij}] \right) = \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=1+i}^n \frac{1}{m} \right) = \\ &= 1 + \frac{1}{mn} \left(\sum_{i=1}^n (n) - \sum_{i=1}^n i \right) = 1 + \frac{1}{mn} \left(n^2 - \frac{n(n+1)}{2} \right) = \\ &= 1 + \frac{n-1}{m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \end{aligned}$$

Відповідно, повний час, необхідний для проведення вдалого пошуку, включаючи час на обрахунок хеш-функції, складає $\Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha)$. ■

Що означає проведений аналіз? Якщо кількість комірок в хеш-таблиці, як мінімум, пропорційна кількості елементів, які зберігаються в ній, то $n = O(m)$ і, відповідно, $\alpha = n/m = O(m)/m = O(1)$, і значить пошук елементу в хеш-таблиці в середньому потребує сталий час. Оскільки в найгіршому випадку вставка елементу в хеш-таблиці займає $O(1)$ часу (як і видалення елементу при використанні двонаправлених списків), можна зробити висновок, що всі словникові операції в хеш-таблиці в середньому виконуються за час $O(1)$.

8.2.4 Уникнення колізій за допомогою відкритої адресації

При використанні методу відкритої адресації всі елементи зберігаються безпосередньо в хеш-таблиці, тобто кожний запис таблиці містить або елемент динамічної множини, або значення NULL. При пошуку елемента ми систематично перевіряємо комірки таблиці до тих пір, доки не знайдемо шуканий елемент або поки не переконаємось у його відсутності в таблиці. Тут, на противагу методу ланцюгів, немає ані списків, ані елементів, які зберігаються поза таблицею. Таким чином, в методі відкритої адресації хеш-таблиця може виявитись заповненою, що унеможливить додавання нових елементів; коефіцієнт заповнення α не може бути більшим за 1.

Для виконання вставки при відкритій адресації ми послідовно перевіряємо комірки хеш-таблиці до тих пір, доки не знайдемо порожню комірку, в яку розміщується новий ключ. Замість фіксованого порядку дослідження комірок 0, 1, ..., $m - 1$ (для чого потрібний час $\Theta(n)$), послідовність досліджуваних комірок залежить від ключа, який вставляється у таблицю. Для визначення досліджуваних комірок хеш-функція розширюється, включаючи в якості другого аргументу номер дослідження (який починається з 0). В результаті хеш-функція стає наступною:

$$h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

У методі відкритої адресації потрібно, що для кожного ключа k послідовність досліджень

$$\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$$

представляла собою перестановку множини $\{0, 1, \dots, m - 1\}$, щоб в кінцевому випадку можна було переглянути всі комірки хеш-таблиці. У наведеному нижче псевдокоді припускається, що елементи в таблиці T представляють собою ключі без додаткової інформації; ключ k тотожній елементу, який містить ключ k . Кожна комірка містить або ключ, або значення NULL (якщо вона не заповнена):

```
HashInsert (T, k)
1           i = 0
```

```

2      repeat j = h(k, i)
3      if T[j] = NIL
4          then T[j] = k
5          return j
6      else i = i + 1
7      until i = m
8      error "Хеш-таблиця переповнена"

```

Лістинг 8.3. Процедура додавання елементу до відкритої адресації.

Алгоритм пошуку ключа k досліджує ту саму послідовність комірок, що алгоритм вставки. Таким чином, якщо при пошуку зустрічається порожня комірка, пошук завершується невдало, оскільки ключ k повинен був бути вставленний в цю комірку у послідовності досліджень, але ніяк не після неї. Процедура HashSearch отримує в якості вхідних параметрів хеш-таблицю T та ключ k і повертає номер комірки, яка містить ключ k (або значення NULL, якщо ключ в хеш-таблиці не знайдений):

```

HashSearch(T, k)
1      i = 0
2      repeat j = h(k, i)
3          if T[j] = k
4              then return j
5          i = i + 1
6      until T[j] = NIL або i = m
7      return NIL

```

Лістинг 8.4. Процедура пошуку ключа у відкритій адресації.

Процедура видалення з хеш-таблиці із відкритою адресацією достатньо складна. При видаленні ключа з комірки i ми не можемо просто помітити її значенням NIL. Зробивши так, ми можемо зробити неможливим вибірку ключа k , в процесі вставки якого досліджувалась і виявилась зайнятою комірка i . Одне з рішень полягає в тому, щоб помічати такі комірки спеціальним значенням DELETED замість NIL. При цьому ми повинні трохи змінити процедуру HashInsert, з тим щоб вона розглядала таку комірку як порожню та могла вставляти в неї новий ключ. В процедурі HashSearch жодних змін робити не потрібно, оскільки ми просто пропускаємо такі комірки при пошуку та досліджуємо наступні комірки під час пошуку в послідовності. Однак при використанні спеціального значення DELETED час пошуку перестає залежати

від коефіцієнту заповнення α , і тому, як правило, при необхідності видалення з хеш-таблиці в якості методу розв'язання колізій обирається метод ланцюгів.

У подальшому ми будемо виходити із припущення **рівномірного хешування**, тобто ми припускаємо, що для кожного ключа в якості послідовності досліджень рівномовірні всі $m!$ перестановок множини $\{0, 1, \dots, m - 1\}$. Рівномірне хешування представляє собою узагальнення визначеного раніше простого рівномірного хешування, яка полягає в тому, що тепер хеш-функція дає не одне значення, а цілу послідовність досліджень. Реалізація істинно рівномірного хешування достатньо важка, але на практиці використовуються прийнятні апроксимації.

Для обчислення послідовності досліджень для відкритої адресації зазвичай використовуються три методи: лінійне дослідження, квадратичне дослідження та подвійне хешування. Ці методи гарантують, що для кожного ключа k $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$ є перестановкою $\{0, 1, \dots, m - 1\}$. Одна ці методи не задовольняють припущенню про рівномірне хешування, адже жоден з них не в змозі згенерувати більше m^2 різних послідовностей (замість $m!$, які потрібні для рівномірного хешування). Найбільша кількість послідовностей досліджень дає подвійне хешування і, як слід, найкращі результати.

Нехай задана звичайна хеш-функція $h': U \rightarrow \{0, 1, \dots, m - 1\}$, яку будемо надалі йменувати **допоміжною хеш-функцією**. Метод лінійного дослідження для обчислення послідовності досліджень використовує хеш-функцію

$$h(k, i) = (h'(k)k + i) \bmod m,$$

де i приймає значення від 0 до $m - 1$. Для заданого ключа k першою досліджуваною коміркою є $T[h'(k)]$, тобто комірка, яку дає допоміжна хеш-функція. Далі досліджуються комірки $T[h'(k) + 1], T[h'(k) + 2], \dots, T[m - 1]$, а потім переходят до $T[0], T[1], \dots, T[h'(k) - 1]$. Оскільки початкова досліджувана комірка однозначно визначає всю послідовність досліджень в цілому, всього є m різних послідовностей.

Лінійне дослідження легко реалізується, однак з ним пов'язана проблема первинної кластеризації, яка полягає у створенні довгих послідовностей

зайнятих комірок, що, саме по собі, збільшує середній час пошуку. Кластери виникають через те, що ймовірність заповнення порожньої комірки, якій передують i заповнених комірок, дорівнює $(i + 1)/m$. Таким чином, довгі серії заповнених комірок мають тенденцію до все більшого подовження, що призводить до збільшення середнього часу пошуку.

Квадратичне дослідження використовує хеш-функцію вигляду

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m,$$

де h' – допоміжна хеш-функція, c_1 та $c_2 \neq 0$ – допоміжні константи, а i приймає значення від 0 до $m - 1$. Початкова досліджувана комірка – $T[h'(k)]$; решта досліджуваних позицій зміщені відносно неї на величину, які описуються квадратичною залежністю від номеру дослідження i . Цей метод працює значно краще лінійного дослідження, але для того, щоб дослідження охопило всі комірки, необхідний вибір спеціальних значень c_1 , c_2 та m ($c_1 = c_2 = 1/2$, $c_1 = c_2 = 1$, $c_1 = 0$, $c_2 = 1$). Окрім того, якщо два ключі мають одну й ту саму початкову позицію дослідження, то однакові й послідовності досліджень у цілому, адже з $h_1(k, 0) = h_2(k, 0)$ слідує $h_1(k, i) = h_2(k, i)$. Ця властивість приводить до більш м'якої вторинної кластеризації. Як і у випадку лінійного дослідження, початкова комірка визначає всю послідовність, тому всього використовуються m різних послідовностей досліджень.

Подвійне хешування представляє собою один з найкращих методів використання відкритої адресації, оскільки отримані при цьому перестановки мають багато характеристик випадкового згенерованих перестановок. Подвійне хешування використовує хеш-функцію наступного вигляду:

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m,$$

де h_1 та h_2 – допоміжні хеш-функції. Початкове дослідження виконується в позиції $T[h_1(k)]$, а зміщення кожної з наступних досліджуваних комірок відносно попередньої дорівнює $h_2(k)$ по модулю m . Відповідно, на відміну від лінійного та квадратичного дослідження, в даному випадку послідовність досліджень залежить від ключа k по двох параметрах – в плані вибору початкової досліджуваної комірки та відстані між сусідніми досліджуваними комірками.

На рис. 8.5 показаний приклад вставки при подвійному хешуванні. Тут наведена хеш-таблиця розміром 13 комірок, в які використовуються допоміжні хеш-функції $h_1(k) = k \bmod 13$ та $h_2(k) = 1 + (k \bmod 11)$. Так як $14 \equiv 1 \pmod{13}$ та $14 \equiv 3 \pmod{11}$, ключ 14 вставляється в порожню комірку 9, після того як при дослідженні комірок 1 та 5 з'ясовується, що ці комірки зайняті.

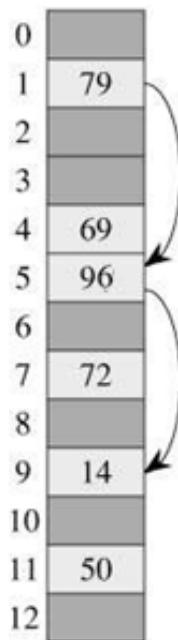


Рис. 8.5. Вставка елементу при подвійному хешуванні.

Для того щоб послідовність досліджень могла охопити всю таблицю, значення $h_2(k)$ повинно бути взаємно простим із розміром хеш-таблиці m . Зручний спосіб забезпечити виконання цієї умови полягає у виборі числа m , рівного ступеню 2, і розробці хеш-функції h_2 таким чином, щоб вона поверталася тільки непарні значення. Ще один спосіб полягає у використанні в якості m простого числа і побудові хеш-функції h_2 такою, щоб вона завжди поверталася натуральні числа, менші m . Наприклад, можна обрати просте число в якості m , а хеш-функції такими:

$$h_1(k) = k \bmod m,$$

$$h_2(k) = 1 + (k \bmod m')$$

,

де m' повинно бути трохи менше m (наприклад, $m - 1$). Скажімо, якщо $k = 123456$, $m = 701$, а $m' = 700$, то $h_1(k) = 80$ та $h_2(k) = 257$, так що першою

досліджуваною коміркою буде 80-а комірка, а потім буде досліджуватись кожна 257-а (за модулем m) комірка, доки не буде знайдена порожня комірка, або доки не виявиться що досліджені всі комірки таблиці.

Подвійне хешування краще лінійного та квадратичного дослідження в сенсі кількості $\Theta(m^2)$ послідовностей досліджень. Це пов'язано з тим, що кожна можлива пара $(h_1(k), h_2(k))$ дає свою, відмінну від інших послідовність досліджень. У результаті продуктивність подвійного хешування достатньо близька до продуктивності «ідеальної» схеми рівномірного хешування.

Аналіз відкритої адресації, як і аналіз методу ланцюгів, виконується із використанням коефіцієнту заповнення $\alpha = n/m$ хеш-таблиці при n та m , які прямують до нескінченності. Зрозуміло, при використанні відкритої адресації $\alpha \leq 1$, адже не може бути більше одного елементу на комірку таблиці ($n \leq m$).

Будемо вважати, що використовуємо рівномірне хешування. За такою ідеалізованою схемою послідовність досліджень $\langle h(k,0), h(k,1), \dots, h(k,m - 1) \rangle$, яка використовується для вставки та пошуку кожного ключа k , з рівною ймовірністю є однією з можливих перестановок множини $\{0, 1, \dots, m - 1\}$. З кожним конкретним ключем пов'язана одна послідовність досліджень, тому під час розгляду розподілу ймовірностей ключем та хеш-функцій всі послідовності досліджень виявляються рівноймовірними.

Теорема 8.3. Математичне сподівання кількості досліджень під час невдалого пошуку в хеш-таблиці з відкритою адресацією та коефіцієнтом заповнення $\alpha < 1$ при припущені рівномірного хешування не більше величини $1/(1 - \alpha)$.

Доведення. При невдалому пошуку кожна послідовність досліджень завершується на порожній комірці. Визначимо випадкову величину X рівну кількості досліджень, які виконуються під час невдалого пошуку, та події A_i ($i = 1, 2, \dots$), які полягають в тому, що було виконано i -е дослідження, і воно прийшлося на заповнену комірку. Тоді подія

$\{X \geq 1\}$ є перетином подій $A_1 \cap A_2 \cap \dots \cap A_{i-1}$. Обмежимо ймовірність $\Pr\{X \geq 1\}$ шляхом обмеження ймовірності $\Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\}$. За узагальненою формулою умовних ймовірностей маємо:

$$\begin{aligned} \Pr\{A_1 \cap A_2 \cap \dots \cap A_{i-1}\} &= \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \dots \\ &\quad \dots \Pr\{A_{i-1} | A_1 \cap A_2 \cap \dots \cap A_{i-2}\}. \end{aligned}$$

Оскільки всього є n елементів та m комірок, $\Pr\{A_1\} = n/m$. Ймовірність того, що буде виконане j -е дослідження ($j > 1$) та воно буде проведено над заповненою коміркою (при цьому перші $j - 1$ досліджені проведені над заповненими комірками), дорівнює $(n - j + 1)/(m - j + 1)$. Ця ймовірність визначається наступним чином: ми повинні перевірити один з $n - (j - 1)$ елементів що лишилися, а всього недосліджених на цей час комірок лишається $m - (j - 1)$. За припущенням рівномірного хешування ймовірність дорівнює відношенню цих величин. Скориставшись тим фактом, що з $n < m$ для всіх $0 \leq j < m$ слідує співвідношення $(n - j)/(m - j) \leq n/m$, для всіх $1 \leq i < n$ отримуємо:

$$\Pr\{X \geq i\} = \frac{n}{m} - \frac{n-1}{m-1} - \frac{n-2}{m-2} \dots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}.$$

Звідси можна обмежити математичне сподівання кількості досліджень:

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=1}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

■

Отримана межа $1 + \alpha + \alpha^2 + \alpha^3 + \dots$ має інтуїтивну інтерпретацію.

Одне дослідження виконується завжди. З ймовірністю, яка близька до α , перше дослідження проводиться над заповненою коміркою, а тому потрібне ще одне дослідження. З ймовірністю, яка близька до α^2 , дві перші комірки виявляються заповненими, а тому потрібне ще одне дослідження, і т.д.

Якщо α – стала, то **теорема 8.3** передбачає, що невдалий пошук виконується за час $O(1)$. Наприклад, якщо хеш-таблиця заповнена наполовину, то середня кількість досліджень при невдалому пошуку буде не більша за $1/(1 - 0,5) = 2$. При наповненості хеш-таблиці на 90% середня кількість досліджень буде не більша за $1/(1 - 0,9) = 10$.

Теорема 8.3 також дає безпосередню оцінку продуктивності процедури `HashInsert`, яка полягає в тому, що середня кількість досліджень при вставці

елементу в хеш-таблицю з урахуванням рівномірного хешування не більша за $1/(1 - \alpha)$.

Теорема 8.4. Математичне сподівання кількості досліджень під час вдалого пошуку в хеш-таблиці з відкритою адресацією та коефіцієнтом заповнення $\alpha < 1$ при припущені рівномірного хешування не більше величини

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}.$$

Доведення. Пошук ключа k виконується в тій самій послідовності досліджень, що й його вставка. Якщо k був $(i + 1)$ -м ключем, який був вставленний в хеш-таблицю, то математичне сподівання кількості досліджень при пошуку k буде не більшим за $1/(1-i/m) = m/(m-i)$. Усереднення по всім n ключам в хеш-таблиці дає нам середню кількість досліджень при вдалому пошуку:

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-1} &= \frac{m}{n} \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha} \left(\frac{1}{m} + \frac{1}{m-1} + \frac{1}{m-2} + \dots + \frac{1}{m-n+1} \right) \\ &= \frac{1}{\alpha} \sum_{k=m-n+1}^m \frac{1}{k} \leq \frac{1}{\alpha} \int_{m-n}^m \left(\frac{1}{x} \right) dx = \frac{1}{\alpha} \ln \frac{m}{m-n} = \frac{1}{\alpha} \ln \frac{m}{1-\alpha} \end{aligned}$$

Якщо хеш-таблиця заповнена наполовину, очікувана кількість досліджень при вдалому пошуку не більша за 1,387; при наповненості в 90% ця величина не перевищує

2,559.

8.2.5 Хеш-функції

Підкатегорії «хеш-функцій»

- загального призначення;
- криптографічні;
- функції формування ключа;
- контрольне число (порівняння);

В рамках нашого курсу будуть розглянуті тільки хеш-функції загального призначення.

8.2.5.1 «Хеш-функції», засновані на діленні

«Хеш-код» як залишок від ділення на число всіх можливих «хешів».

Хеш-функція може обчислювати «хеш» як залишок від ділення вхідних даних на M:

$$h(k) = k \bmod M$$

де M – кількість всіх можливих «хешів» (вихідних даних).

При цьому очевидно, що при парному M значення функції буде парним при парному k і непарних - при непарному k. Також не слід використовувати в якості M ступінь основи системи числення комп'ютер, так як «хеш-код» буде залежати тільки від декількох цифр числа k, розташованих праворуч, що призведе до великої кількості колізій. На практиці зазвичай вибирають просте M; в більшості випадків цей вибір цілком задовільний.

Приклад. Маємо наступну послідовність чисел:

№ п/п	1	2	3	4	5	6	7	8	9	10	11	12
Послідовність	2	5	8	9	12	16	19	20	23	25	27	35

$$h(k) = k \bmod 11$$

Хеш-таблиця

№	хеш	ланцюг	індекси
1	1	12 → 23	5, 9
2	2	2 → 35	1, 12
3	3	25 →	10
4	4	→	
5	5	5 → 16 → 27	2, 6, 11
6	6	→	
7	7	→	
8	8	8 → 19	3, 7
9	9	9 → 20	4, 9
10	10	→	
11	0	→	

8.2.5.2 «Хеш-функції», що базуються на множенні

Позначимо символом w кількість чисел, які представлені машинним словом. Наприклад, для 32-роздрядних комп'ютерів, сумісних з IBM PC, $w = 2^{32}$.

Виберемо деяку константу A так, щоб A була взаємно простою з w . Тоді хеш-функція, яка використовує множення, може мати наступний вигляд:

$$h(K) = \left[M \left\lfloor \frac{A}{w} * K \right\rfloor \right]$$

У цьому випадку на комп'ютері з двійковою системою числення M є ступенем двійки, і $h(K)$ буде складатися з старших бітів правої половини добутку $A * K$.

Серед переваг хеш-функцій, заснованих на діленні і множенні, варто відзначити вигідне використання невипадковості реальних ключів. Наприклад, якщо ключі являють собою арифметичну прогресію (наприклад, послідовність імен «Ім'я 1», «Ім'я 2», «Ім'я 3»), хеш-функція, яка використовує множення, відобразить арифметичну прогресію в наближену арифметичну прогресію різних хеш-значень, що зменшить кількість колізій в порівнянні з випадковою ситуацією.

8.2.5.3 Хеш-функції загального призначення

FNV (англ. Fowler–Noll–Vo) – проста хеш-функція для загального призначення, розроблена Гленом Фаулером, Лондоном Керт Нолом і Фогном Во. Не є криптографічною функцією. Існують варіанти для 32-, 64-, 128-, 256-, 512-, і 1024-бітових хешів.

Функція проста в реалізації. Її основа - множення на просте число і складання по модулю 2 з вхідним текстом.

Псевдокод:

```
const unsigned FNV_32_PRIME = 0x01000193; //16777619 простое число

unsigned int FNV1Hash (char *buf)
{
    unsigned int hval = 0x811c9dc5; // FNV0 hval = 0 //2166136261
```

```

while (*buf)
{
    hval *= FNV_32_PRIME;
    hval ^= (unsigned int)*buf++;
}

return hval;
}

```

MurmurHash2 - проста і швидка хеш-функція загального призначення, розроблена Остіном Епллбі (2008 рік). Не є криптографічно-безпечною, повертає 32-роздрядне беззнакове число. З переваг функції автором відзначена простота, хороший розподіл, потужний лавинний ефект, висока швидкість і порівняно висока стійкість до колізій. Поточні версії алгоритму оптимізовані під Intel-сумісні процесори.

Друга версія хеш-функції має деякі недоліки. Зокрема, це проблема колізій на невеликих рядках. Виправлений варіант має структуру типу Merkle-Damgard, виконується трохи повільніше (приблизно на 20%), але показує кращу статистику.

Лавинний ефект (англ. Avalanche effect) - поняття в криптографії, зазвичай застосовується до блокових шифрів і криптографічних хеш-функцій. Важлива криптографічна властивість для шифрування, яка означає, що зміна значення малої кількості бітів у вхідному тексті або в ключі веде до «лавинної» зміни значень вихідних бітів шифротексту. Іншими словами, це залежність всіх вихідних бітів від кожного вхідного біта.

MurmurHash2

```

unsigned int MurmurHash2 (char * key, unsigned int len)
{
    const unsigned int m = 0x5bd1e995;
    const unsigned int seed = 0;
    const int r = 24;

    unsigned int h = seed ^ len;

    const unsigned char * data = (const unsigned char *)key;
    unsigned int k;

```

```

while (len >= 4)
{
    k = data[0];
    k |= data[1] << 8;
    k |= data[2] << 16;
    k |= data[3] << 24;

    k *= m;
    k ^= k >> r;
    k *= m;

    h *= m;
    h ^= k;

    data += 4;
    len -= 4;
}

switch (len)
{
    case 3:
        h ^= data[2] << 16;
    case 2:
        h ^= data[1] << 8;
    case 1:
        h ^= data[0];
        h *= m;
};

h ^= h >> 13;
h *= m;
h ^= h >> 15;

return h;
}

```

MurmurHash2A

```

#define mmix(h,k) { k *= m; k ^= k >> r; k *= m; h *= m; h ^= k; }

unsigned int MurmurHash2A ( const void * key, int len, unsigned int seed )
{

```

```
const unsigned int m = 0x5bd1e995;
const int r = 24;
unsigned int l = len;

const unsigned char * data = (const unsigned char *)key;

unsigned int h = seed;
unsigned int k;

while(len >= 4)
{
    k = *(unsigned int*)data;

    mmix(h,k);

    data += 4;
    len -= 4;
}

unsigned int t = 0;

switch(len)
{
case 3: t ^= data[2] << 16;
case 2: t ^= data[1] << 8;
case 1: t ^= data[0];
};

mmix(h,t);
mmix(h,l);

h ^= h >> 13;
h *= m;
h ^= h >> 15;

return h;
}
```