МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний технічний університет
«Харківський політехнічний інститут»

МЕТОДИЧНІ ВКАЗІВКИ
до виконання самостійних робіт
з курсів: «Інформаційні технології і програмування»
«Інформаційні процеси в телекомунікаційних системах»

МЕТОДИ УПОРЯДКУВАННЯ ДАНИХ

для студентів спеціальностей:
151 Автоматизація та комп'ютерно-інтегровані технології
172 Телекомунікації та радіотехніка

затверджено
редакційно-видавничою
радою університету,
протокол №3 від 26.10.2022 р.

Харків
НТУ «ХПІ»
2022

Методичні вказівки для виконання самостійних робіт «**Методи упорядкування даних**» для студентів спеціальностей 151 Автоматизація та комп'ютерно-інтегровані технології, 172 Телекомунікації та радіотехніка / Уклад. А.О. Зуєв, Д.Г. Караман, Д.А. Гапон - Х.: НТУ «ХПІ», 2022. - 48 с.

Укладачі: А.О. Зуєв
Д.Г. Караман
Д.А. Гапон

Рецензент О.Є. Тверитникова

Кафедра інформаційно-вимірювальних технологій і систем

ВСТУП

Теорія графів - один із найбільших розділів дискретної математики, широко застосовується у розв’язанні економічних та управлінських завдань, у програмуванні, хімії, конструюванні та вивченні електричних ланцюгів, комунікацій, психології, соціології, лінгвістиці, та інших галузях знань. Теорія графів систематично і послідовно вивчає властивості графів, про які можна сказати, що вони складаються з множини точок і множини ліній, що відображають зв'язки між цими точками. Засновником теорії графів вважається Леонард Ейлер.

Упорядкування даних є найважливішим науково-практичним завданням з часів появи комп'ютерів. Термін кластерний аналіз, було введено в 1939 році. Він включає в себе більше ніж сотні різноманітних алгоритмів, які відповідають на загальне питання, що виникає в різних областях людської діяльності – як організувати отримані дані в наочні структури.

У середині XX століття також відбувся інтенсивний розвиток теорії сортування. Було запропоновано безліч різних алгоритмів сортування: злиття зі вставкою, обмінне порозрядне сортування, каскадне злиття та метод Шелла, багатофазне злиття та швидке сортування Хоара, пірамідальне сортування Вільямса та обмінне сортування злиттям Бетчера. Алгоритми, що з'явилися пізніше, багато в чому були варіаціями вже відомих методів. Набули поширення адаптивні методи сортування, орієнтовані більш швидке виконання у випадках, коли вхідна послідовність задовольняє заздалегідь встановленим критеріям.

Методичні вказівки призначені для вивчення основ теорії графів, алгоритмів сортування та кластерного аналізу. Розглянуто основні алгоритми упорядкування даних та роботи з графами, а також алгоритми кластерного аналізу. Для всіх алгоритмів розглянуті приклади реалізації на мові C++.

Також в методичних вказівках представлені індивідуальні завдання для виконання лабораторних і практичних робіт, які допоможуть ефективному освоєнню програмування та інформаційних технологій.

1 АЛГОРИТМИ НА ГРАФАХ

Теорія графів є основою сучасної прикладної математики. Універсальність графів незамінна при проектуванні та аналізі різноманітних комунікаційних мереж. Теорія графів, як математичне знаряддя, застосовується як до практичних наук (теорія інформації, теорія систем, проектування транспортних мереж), так і до чисто абстрактних дисциплін.

У різних сферах знань поняття граф може зустрічатися як:

- структура;
- мережа;
- соціограма;
- молекулярна структура;
- навігаційна карта;
- розподільна мережа.

Теорія графів як множина методів та алгоритмів які працюють з системою ліній, що з'єднують точки, дуже зручна, тому що:

- має геометричну наочність;
- має математичну змістовність;
- немає громіздкого математичного апарату.

Граф – це сукупність об'єктів із зв'язками між ними. Об'єкти розглядаються як вершини, або вузли графу, а зв'язки – як ребра.

Граф – множина вершин V і множина ребер E, що з'єднують пари різних вершин. Одне ребро містить максимум одну пару вершин.

Для різних застосувань види графів можуть відрізнятися орієнтованістю, обмеженнями на кількість зв’язків і додатковими даними про вершини або ребра. Ребра графу можуть бути направленими або не направленими і також можуть мати вагу.

Геометричне представлення графу – фігура на площині, яка складається з непорожньої скінченної множини V вершин і скінченної множини E орієнтованих чи неорієнтованих ліній (ребер), що з'єднують деякі пари вершин.

Шлях – перехід між цільовою та вихідною вершинами графа по ребрам, де кожне з ребер використовується лише один раз.

Зв'язний граф – граф, у якому між будь-якою парою вершин існує шлях.

Ребро, що сполучає вершину саму зі собою називається **петлею**.

Цикл – замкнутий ланцюг (шлях) ребер на графі, для орієнтованих графів цикл називається **контуром**.

Дерево – зв'язний граф без циклів та петель.

Приклад графів — географічні карти, гіпертекст, мікросхеми, розклад, транзакції телефонної компанії, обчислювальні мережі, структури програм.

Граф може бути заданий матрицею суміжності або списком суміжних вершин.

1.1 Типи графів і їх частини

Граф, у якого є дублюючі ребра або ребра, що з'єднують одні й ті ж вершини (петлі), називається **мультиграф**. Якщо в графі немає петель та дублюючих ребер, такий граф називається **простий граф**.

Простий граф, що складається з V вершин, містить не більше ніж E = V(V-1)/2 ребер.

Суміжні вершини — вершини з'єднані ребром. **Ступінь вершини** — кількість ребер, що виходять з цієї вершини (інцидентних ребер).

Підграф — граф, що є підмножиною графа і сам є графом.

Планарний граф — граф без перетину ребер на площині.

Евклідів граф — такий граф в якому є масштаб і необхідно витримувати відстань між вершинами. Наприклад, географічна карта.

Шлях — послідовність суміжних вершин, в якій всі вершини різні.

Цикл — шлях, у якого однакові початкова і кінцева вершини. Якщо в графі немає циклів, він називається **ациклічним**.

Контур — такий цикл, що включає всі вершини графа.

Зважений граф, такий граф, в якому кожне ребро має певну вагу.

Довжина шляху (циклу) — кількість ребер, що його складають або сума їх ваг.

Непересічні шляхи — такі шляхи, у яких немає спільних вершин, крім кінцевих точок.

Пов'язаний граф — такий граф, в якому існує шлях з кожної вершини в будь-яку іншу.

Дерево — ациклічний пов'язаний граф.

Ліс — множина дерев.

Кістяк — підграф, що містить всі вершини графа, який являє собою єдине дерево.

Кістяковий ліс — підграф, що містить всі вершини графа і є лісом.

Графи, у яких присутні всі ребра, називаються **повними**. **Кліка** — повний підграф.

Насиченість графа: a = 2E/V.

Орієнтований граф (орграф) — такий граф, в якому ребра односпрямовані.

1.2 Представлення графа

У зв'язку з широким застосуванням графів в програмуванні та інформаційних технологіях взагалі, виникає питання про представлення графа у вигляді структури даних. Різні способи представлення графів в пам'яті комп'ютера відрізняються обсягом зайнятої пам'яті та швидкістю виконання різних операцій над графом.

Найбільш часто використовуються наступні три структури даних: матриця суміжності, матриця та список інцидентності.

Інцидентність вершини *a* до вершини *b* означає, що вершина *a* є початком, а вершина *b* є кінцем ребра. Таким чином, дві вершини називаються інцидентними, якщо у них є спільне ребро і можливо перейти з одної вершини до другої.

Матриця суміжності — це квадратна матриця S, в якій і число рядків, і число стовпців дорівнює V — числу вершин графа. В кожну комірку Sij матриці суміжності записуються числа не рівні 0, якщо вершини і та j з'єднані ребром, тобто є суміжними.

Для задавання матриці суміжності зазвичай використовується двовимірний масив розміром V×V, де V — число вершин графа.

Матриця суміжності для неорієнтованого графа. Елемент матриці суміжності Sij неорієнтованого графа визначається наступним чином:

а) 1, якщо вершини і та j суміжні;

б) 0, якщо вершини і та j не суміжні.

Якщо Sij = 1 та i = j, тобто елемент знаходиться на діагоналі і цей елемент дорівнює одиниці, то ця вершина має петлю (вершина графа «суміжна сама до себе»). Якщо ж Sij = 0, то вершина не має петлі.

Матриця суміжності для орієнтованого графа. Для орієнтованого графа необхідно не тільки визначати наявність ребра, а й його направленість. Тому елемент матриці суміжності Sij орієнтованого графа визначається наступним чином:

а) 1, якщо з вершини і до вершини j входить ребро;

б) 0, якщо з вершини і до вершини j ребро не входить.

Як і для неорієнтованих графів, якщо для елемента матриці Sij = 1, де i = j, тобто елемент знаходиться на діагоналі і дорівнює одиниці, означає що ця вершина має петлю.

Щоб визначити **ступінь вершини** необхідно порахувати суму одиниць в рядку матриці суміжності.

Матриця інцидентності будується за схожим принципом, як і для матриці суміжності. Але, якщо остання має розмір V×V, де V — число вершин, то матриця інцидентності визначається розміром V×E, де E — кількість ребер. Щоб задати значення для будь якого елементу матриці, необхідно зіставити не вершину з вершиною, а вершину з ребром.

У кожну клітинку матриці інцидентності неорієнтованого графа записується 0 або 1, а в разі орієнтованого графа: 1, 0 або -1:

Неорієнтований граф:

а) 1 — вершина інцидентна ребру;

б) 0 — вершина не інцидентна ребру.

Орієнтований граф:

а) 1 — вершина інцидентна ребру, і є його початком (ребро виходить з вершини);

б) 0 — вершина не інцидентна ребру;

в) -1 — вершина інцидентна ребру, і є його кінцем (ребро заходить у вершину).

Матриця суміжності, як і матриця інцидентності, дозволяє встановити множину вершин, сусідніх до заданої, не вдаючись до повного перегляду всього графу.

1.3 Пошук шляху між двома елементами

Простий шлях — будь-який шлях між двома вершинами.

```
bool check_simple_path( int v, int w, bool* visited)
{
    if( v == w )
        return true;
    visited[v] = true;
    int n = 0, a = adjacent( v, n );
    while( a >= 0 )
    {
        if( !visited[r] && check_simple_path( r, w , visited ) )
            return true;
        a = adjacent( v, n );

    }// while( a >= 0 )

    return false;
}
```

Функція adjacent(v, n) повертає суміжну вершину з індексом n для вершини v, або -1 у випадку якщо немає такої вершини. Масив visited має розмірність рівну кількості вершин і повинен бути заповнений значеннями false.

Гамильтонов шлях — простий шлях, що проходить через всі вершини графа один раз.

```
bool check_hamilt_search( int v, int w, int d, bool* visited )
{
    if(v == w ) return d == 0;
    visited[v] = true;
    int n = 0, a = adjacent( v, n );
    while( a >= 0 )
    {
        if( !visited[r] && check_hamilt_search ( r, w, d-1, visited ) )
            return true;
        a = adjastent( v, n );

    }// while( a >= 0 )
    visited[v] = false;
    return false;
}
```

Складність цього алгоритму O(V!), де V — кількість вершин у графі. Припустимо що граф з 15 вершинами обробляється за 1 секунду, тоді граф з 19 вершинами буде оброблятися за добу, а з 21 вершиною — шість століть.

Шлях (цикл) Ейлера — простий шлях, що проходить через кожне ребро один раз.

Відомо, що в графі існує цикл Ейлера, коли він пов'язаний і всі його вершини мають парну ступінь. Визначення наявності шляху Ейлера — перевірка парності ступенів вершин через масив ступенів deg, кожний елемент котрого містить суму не рівних 0 елементів кожного рядку матриці суміжності.

```
bool check_even( int v, int w, const int* deg )
{
    int t = deg[v] + deg[w];
    if( t % 2 ) return false;

    for( int t = 0; t < V; ++t )
    {
        if( deg [t] % 2 )
            return false;
    }
    return true;
}
```

Якщо умови наявності шляху Ейлера виконуються (граф пов'язаний і ступеня всіх його вершин парні), то можна застосувати алгоритм пошуку циклу Ейлера за методом видалення циклів (рис.1.1).

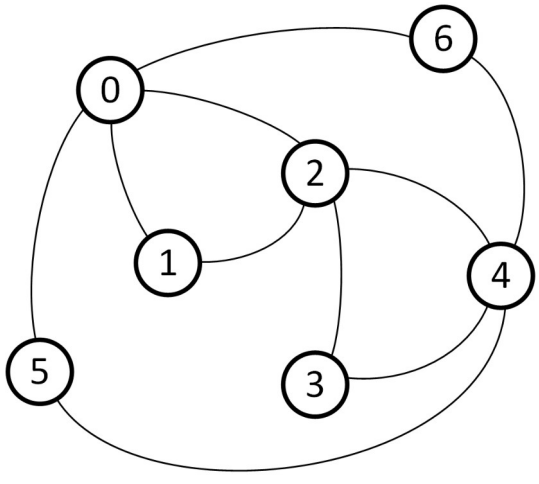

Рис.1.1 – Приклад пов'язаного графу для пошуку циклу Ейлера

Пройдемо по циклічному шляху, стираючи все ребра і додаючи в стек всі вершини, які зустрінуться, так, аби можна було простежити шлях, отримати всі його

ребра. Потрібно переглянути кожну вершину на наявність бічних шляхів, які можуть бути включені в головний шлях.

Приклад для пошуку циклу Ейлера:

- порахувати ступеня вершин;
- знайти цикл Ейлера.

1) Визначимо ступені вершин:

V0 {4}, V1 {2}, V2 {4}, V3 {2}, V4 {4}, V5 {2}, V6 {2}.

Всі ступені вершин парні, граф пов'язаний — це означає, що цикл існує і можна переходити до його пошуку.

2) Пошук циклу Ейлера методом видалення циклів.

Почнемо пошук з вершини 0:

V0-V6, V6-V4, V4-V5, V5-V0.

Після цих кроків зовнішній контур буде повністю видалено, і пошук повернувся до вершини 0.

Стек: **0, 6, 4, 5, 0**

Отримані ребра: 0-6, 6-4, 4-5, 5-0.

Продовжуємо пошук з вершини 0:

V0-V2, V2-V4, V4-V3, V2-V1, V1-V0

Стек: 0, 6, 4, 5, 0, **2, 4, 3, 2, 1, 0**

Таким чином, видалено всі ребра і пошук повернувся у початкову вершину. Нескладно переконатися, що цикл буде успішно знайдено і в тому випадку, якщо пошук буде розпочато з будь-якої вершини і при будь-якому виборі ребра що видаляється в кожній вершині.

Підсумковий цикл буде отримано зі стеку вершин, шляхом складання з них ребер циклу: 0-1, 1-2, 2-3, 3-4, 5-2, 2-0, 0-5, 5-4, 4-6 , 6-0

1.4 Властивості дерев

Дерева і алгоритми що пов'язані з ними є дуже важливою частиною теорії графів і застосовуються в багатьох структурах упорядкування даних.

Нехай G — неорієнтований граф, тоді такі затвердження еквівалентні:

1) G — дерево;

2) G — пов'язаний граф і E = V - 1;

3) G — ациклічний граф;

4) будь-яку пару вершин пов'язує тільки одне ребро;

5) G не має циклів, але при додаванні довільного ребра в ньому виникає рівно один цикл.

1.5 Пошук мінімального кістякового дерева

Однією з найважливіших завдань пов'язаних з деревами, є завдання пошуку мінімального кістякового дерева — такого дерева, сума ваг ребер якого була б мінімальною для графа. На практиці, завдання пошуку мінімального кістякового дерева виникає в багатьох випадках, наприклад, при проектуванні доріг, електромереж, трубопроводів і т.д. У ситуаціях, коли необхідно пов'язати множину об'єктів комунікаційними лініями, сумарна вартість яких повинна бути мінімальною.

Підграф T неорієнтованого графа G = (V, E) називається **кістяком** (кістяковим деревом), якщо T є деревом і множина його вершин збігається з V. Вагою кістякового дерева називають суму ваг всіх його ребер.

Алгоритм Крускала для визначення мінімального кістякового дерева T для графа G(V, E), полягає в наступному: вирізається ребро з найменшою вагою з початкового графа, і, якщо воно не утворює циклу в кістяковому дереві, воно додається в нього. Побудова дерева триває доти, поки у початковому графі є хоча б одне ребро.

Таким чином, алгоритм забезпечує сортування всіх ребер в порядку зростання довжини, і додавання їх по черзі в мінімальний кістяк, якщо вони з'єднують різні компоненти зв'язності. Цю умову можна визначити, наприклад, за допомогою спеціальної структури даних — системи непересічних множин (англ. Disjoint Set Union).

Кількість ребер в графі G є n, тоді, за алгоритмом Крускала:

1) вважаємо T(0) = G(V, 0);

2) для i = 0, .. n - 1 вважаємо T(i + 1) = T(i) + e, де e — ребро з графу G з мінімальною вагою, таке, що не є ребром T(i) та не утворює циклу з ребрами T(i).

Система непересічних множин (DSU) — структура даних для ефективної роботи з непересічними множинами (в таких множинах кожен елемент належить тільки до однієї множини), що дозволяє перевіряти належність пари елементів до однакової множини, і об'єднувати множини.

DSU є набором кореневих дерев (ліс). Кожне дерево відповідає певній множині. При використанні DSU необхідно лише підніматися вгору по деревах, тому досить зберігати для кожної вершини тільки номер її безпосереднього предка. Для цього використовується спеціальний масив p[V].

Для опису множини використовується номер вершини, що є коренем відповідного дерева. Тому для визначення, чи належать два елементи до однієї множини, потрібно для кожного елемента знайти корінь відповідного дерева (піднімаючись вгору поки це можливо) і порівняти ці корні.

Якщо потрібно об'єднати множини з корінням *a* та *b*, дамо p[*a*] = *b*, тим самим все дерево а буде підвішене до кореня дерева *b*.

Ще один алгоритм, який застосовується для побудови кістякового дерева, це **алгоритм Прима.** Він заснований на твердженні, що, якщо розділити вершини графа на дві множини: оброблені та необроблені, перша з яких становить зв'язну частину мінімального кістякового дерева, то ребро мінімальної довжини, що зв'язує ці дві множини, гарантовано буде входити до мінімального кістяку.

1) Вважаємо T(0) = (V1, E1), де V1 = {a, b}, E1 = {a, b}, і вага ребра ab — мінімальна у множині E.

2) Для i = 1, ..., n-1 вважаємо T(i + 1) = T(i) + e, де e — ребро графу G з мінімальною вагою таке, що:

- не є ребром T(i);
- пов'язує ребро T(i) з вершиною G, що не належить T(i).

Обидва алгоритми мають складність O(n log m), але існують константні відмінності в швидкодії їх роботи. На розріджених графах, де кількість ребер

приблизно дорівнює кількості вершин, швидше працює алгоритм Крускала, а на насичених (кількість ребер приблизно дорівнює квадрату кількості вершин) — алгоритм Прима, при використанні матриці суміжності. На практиці частіше використовується алгоритм Крускала.

1.6 Топологічне сортування

Топологічне сортування — упорядкування вершин безконтурного орієнтованого графа у частковому порядку, який задано ребрами орграфа на множині його вершин.

Безконтурний орієнтований граф також називають орієнтованою мережею (або просто мережею).

Припустимо для деякої фірми потрібно виконати n проектів, для яких задано відношення порядку i → j, тобто проект j не може початися раніше, ніж закінчиться проект i. На практиці, це означає що, проект j використовує результати реалізації проекту i. Необхідно знайти припустимий порядок проходження проектів в часі.

Нехай порядок проходження заданий у вигляді орграфа G = (V, E). Ребро (i, j) належить до множини E тільки тоді, коли правдиве відношення i → j. Розв'язок задачі існує тільки в тому випадку, якщо отриманий орграф не містить циклів. Якщо цикл існує, то жоден з проектів, що входять до циклу, не можна починати. Нехай перестановка (i_1, ..., i_n) визначає новий порядок виконання проектів. Цей порядок є допустимим, якщо звідношення $i_j \rightarrow i_k$ слідує, що i_j передує i_k в отриманій перестановці.

Введемо номер вершини label(i) що належить {1 ... n}. Таким чином, задача зводиться до відшукання такої нумерації вершин графа G, щоб для будь якого ребра (i, j), яке належить E, виконувалась умова label(i) < label(j). Послідовність проектів в порядку зростання їх номерів і буде розв'язком задачі. Відшуканням такої нумерації займаються алгоритми **топологічного сортування** вершин графа.

Існує кілька способів топологічного сортування, із найбільш відомих:

– алгоритм Демукрона;

– метод сортування за допомогою представлення графа у вигляді декількох рівнів;

– метод топологічного сортування за допомогою обходу в глибину.

Припустимо, що граф задано списками безпосередніх послідовників E0(i), $i = 1 \ldots n$. Крім того, задано кількість безпосередніх послідовників b_i для кожної вершини. У контексті прикладу, що розглядається, **безпосередній послідовник** проекту i — такий проект j, який може початися відразу після закінчення i, але ніяк не раніше.

1) задаємо поточний номер j = 1;

2) переглядаємо список b_i та створюємо список вершин J, у яких немає попередників, тобто $b_i = 0$; якщо J залишається порожнім, то в графі є цикл і розв'язок задачі неможливий;

3) надаємо поточний номер j будь-якій вершині зі списку J, розглядаємо всіх її безпосередніх послідовників і таких, що належать до E0(j) та перетворюємо $b_i = b_i - 1$; якщо $b_i = 0$, то вершину i включаємо до J;

4) після того, як множину безпосередніх послідовників E0(j) переглянуто, видаляємо j з J; якщо поточний номер не дорівнює n, збільшуємо його на одиницю. Повторюємо крок 3, якщо список J все ще не порожній.

Складність алгоритмів топологічного сортування становить O(n+m), за операціями і O(n) за витратами пам'яті. Топологічне сортування застосовується в найрізноманітніших випадках, наприклад, при розпаралелюванні алгоритмів, коли за деяким описом алгоритму потрібно скласти граф залежностей його операцій, та відсортувавши його топологічно, визначити, які з операцій є незалежними і можуть виконуватися паралельно (одночасно). Прикладом використання топологічного сортування також може бути створення карти сайту, у якого є деревоподібна система розділів.

1.7 Найкоротші шляхи

Ще одне, важливе завдання, яке вирішується за допомогою графів — пошук найкоротшого шляху. **Довжиною шляху** вважається сума ваг ребер, які входять в шлях, а шлях мінімальної довжини з вершини i до вершини j називається **найкоротшим шляхом**. Всі алгоритми пошуку найкоротшого шляху можна розділити на дві групи: пошук шляху з однієї вершини в усі інші, та пошук всіх шляхів з усіх вершин в усі інші — пошук k-найкоротших шляхів (так звані матричні методи).

1) Алгоритм Флойда-Уоршелла. Алгоритм знаходить шлях від кожної вершини до кожної іншої з обчислювальною складністю $O(n^3)$, де n — кількість вершин у графі. Ваги ребер графа можуть бути негативними, але не може бути циклів з негативною сумою ваг ребер.

У двовимірному масиві d[n][n] на ітерації *i* буде зберігатися розв’язок вихідної задачі з обмеженням на те, що в якості "пересадних" у шляху будуть використовуватися вершини з номером строго менше за $i - 1$. Нехай йде ітерація *i*, та необхідно оновити масив шляхів до ітерації $i + 1$. Для цього, для кожної пари вершин обирається в якості "пересадної" вершина $i - 1$, та, якщо цей вибір покращує результат, такий стан зберігається на наступну ітерацію. Всього необхідно зробити n + 1 ітерацію, після її завершення в якості "пересадної" можна використовувати будь-яку вершину, а масив d буде розв’язком завдання.

```
// g[n][n] - масив, в якому зберігаються ваги ребер,
// g[i][j] = ∞, якщо ребра між i та j немає
d = g;
for( int i = 1; i < n+1; ++i )
{
    for( int j = 0; j < n; ++j )
    {
        for( int k = 0; k < n; ++k )
        {
            if( d[j][k] > d[j][i - 1] + d[i - 1][k] )
            d[j][k] = d[j][i - 1] + d[i - 1][k];
        }
    }
```

```
}
```

2) Алгоритм Форда-Беллмана. Алгоритм дозволяє знайти відстань від однієї вершини (з номером 0) до всіх інших з обчислювальною складністю O(n×m), де m — кількість ребер. Аналогічно з попереднім алгоритмом, ваги можуть бути негативними, але не може бути циклів з негативною сумою ваг ребер.

Заведемо масив d[n], в якому на ітерації *i* буде зберігатися розв'язок вихідної задачі з обмеженням на те, що в шлях має входити строго менше ніж *i* ребер. Якщо таких шляхів до вершини j немає, то d[j] = ∞. На самому початку масив d повинен бути заповнений ∞. Щоб оновити масив на ітерації *i*, необхідно пройти по кожному ребру і спробувати поліпшити відстань до вершин, які поєднано даним шляхом. Найкоротші шляхи не містять циклів, і, оскільки всі цикли невід'ємні, можна прибрати цикл з шляху, при цьому довжина шляху не погіршиться. Необхідно також відзначити, що саме так можна знайти цикли з негативною вагою в графі: необхідно зробити ще одну ітерацію і подивитися, чи не поліпшилася відстань до якої-небудь вершини. Довжина найкоротшого шляху буде не більше за n – 1 ребер, тому, після ітерації n, масив d буде містити розв'язок задачі.

```
// e[m] - масив, в якому зберігаються ребра та їх ваги
// first, second - вершини, що з'єднуються ребром, value - вага ребра

for( int i = 0; i < n; ++i ) d[i] = ∞;

d[0] = 0;
for( int i = 1; i <= n; ++i )
{
    for( int j = 0; j < m; ++j )
    {
        if( d[e[j].second] > d[e[j].first] + e[j].value )
            d[e[j].second] = d[e[j].first] + e[j].value;
        if( d[e[j].first] > d[e[j].second] + e[j].value )
            d[e[j].first] = d[e[j].second] + e[j].value;
    }
}
```

3) Алгоритм Дейкстри. Алгоритм використовується для побудови найкоротшого шляху з вершини s в усі інші вершини графа зі складністю $O(n^2)$. Всі ваги графа повинні бути невід'ємні.

На кожній ітерації алгоритму треба буде маркувати деякі вершини, для цього необхідно завести два масиви: mark[n], в якому буде міститися true, якщо вершина позначена, і false, якщо непозначена; d[n], в якому для кожної вершини буде зберігатися довжина найкоротшого шляху, що проходить тільки по маркованим вершинам, що використовуються як "пересадні". Для маркованих вершин, довжина, зазначена в d, і є розв'язком задачі. Спочатку маркується тільки вершина s. Елемент g[i] дорівнює:

– *x*, якщо вершини s та *i* з'єднує ребро з вагою *x*;

– ∞, якщо їх не з'єднує ребро;

– 0, якщо *i* = s.

На кожній ітерації знаходиться вершина v з найменшим значенням в d серед всіх маркованих. Тоді значення d[v] є розв’язком для v. Припустимо, найкоротший шлях до v з s проходить не тільки по маркованим вершинам, які виконують роль "пересадних", і при цьому він коротше ніж d[v]. Якщо взяти першу немарковану вершину u, яка зустрінеться на цьому шляху, то довжина пройденої частини шляху (від s до u) дорівнює d[u], а len >= d[u], де len — довжина найкоротшого шляху з s до v (тому що від'ємних ребер немає). Оскільки передбачається, що len менша за d[v], то d[v] > len >= d[u]. Але тоді v не відповідає визначенню — у неї не найменше значення d[v] серед вершин, які марковані.

Маркуємо вершину v та перераховуємо масив d до тих пір, поки всі вершини не стануть маркованими, тоді в d буде рішення задачі.

```
// g[n][n] - масив, в якому зберігаються ваги ребер,
// g[i][j] = ∞, якщо ребра між i та j немає

d = g;
d[0] = 0;
for( int i = 0; i < n; ++i ) mark[i] = ( i == 0 );

for( int i = 1; i < n; ++i )
{
    v = -1;
    for( int j = 0; j < n; ++j )
    {
        if( !mark[j] && ( v == -1 || d[v] > d[i] )
            v = j;
    }
    mark[v] = true;

    for( int j = 0; j < n; ++j )
    {
        if( d[i] > d[v] + g[v][i] )
            d[i] = d[v] + g[v][i];
    }
}
```

4) Алгоритм Дейкстри для розріджених графів. Цей алгоритм вирішує завдання, аналогічне до попереднього, але має складність в середньому O(m×log(n)). Слід зауважити, що m може бути порядку n^2, тобто цей варіант алгоритму Дейкстри не завжди швидше попереднього, а тільки при досить маленьких m.

Розглянемо, що необхідно для реалізації алгоритму Дейкстри. Необхідно знаходити за значенням d мінімальну вершину *i* оновлювати значення d для довільної

вершини. У попередній реалізації використовувався масив, в якому знайти мінімальну за значенням d вершину можна зі складністю O(n), а оновити — зі складністю O(1). Для прискорення операції пошуку використаємо спеціалізовану структуру даних — бінарну купу, яка підтримує операції:

- додати в купу елемент (складність O(log(n)));
- знайти мінімальний елемент (складність O(1));
- видалити мінімальний елемент (складність O(log(n)), де n — кількість елементів в купі).

Створимо масив d[n], аналогічно з попереднім методом, і купу q. У купі будуть зберігатися пари {v, d[v]} (v — номер вершини), для порівняння буде використовуватися d[v]. Також в купі можуть бути і фіктивні елементи, які з'являються, тому що значення d[v] оновлюється, але його не можна змінити в купі. Тому в купі можуть бути кілька елементів з однаковим номером вершини *i* з різним значенням d, але загалом вершин в купі буде не більше m. Коли вибирається мінімальне значення з купи, необхідно перевірити, чи не є цей елемент фіктивним. Для цього достатньо порівняти значення d в купі і реальне його значення. Для запису графа замість масиву доцільніше використовувати масив списків.

```
// g[n] - масив списків, в кожному списку зберігаються пари:
// first - вершина, поєднана з вершиною і ребром, second - вага цього ребра

d[0] = 0;
for( int i = 0; i < n; ++i ) d[i] = ∞;

for( e in g[0] )
{
    d[e.first] = e.second;
    q.add( pair( e.second, e.first ) );
}
for( int i = 1; i < n; ++i )
{
    v = -1;
    while( v == -1 || d[v] != val )
    {
        v = q.top().second;
        val = q.top().first;
        q.removeTop();
    }
    mark[v] = true;
    for( e in g[v] )
    {
        if( d[e.first] > d[v] + e.second )
        {
            d[e.first] = d[v] + e.second;
```

```
            q.add( pair( d[e.first], e.first ) );
        }
    }
}
```

5) Метод Шимбелла. Метод використовується для зваженого орграфа з від'ємними вагами. Цей метод відноситься до групи матричних і дозволяє отримати всі можливі шляхи з усіх вершин. Алгоритм дає найкоротші шляхи між усіма парами вузлів графа, за умови, що в графі немає контурів.

Суть алгоритму полягає в зведенні в ступінь r матриці ваг ребер $L = ||l_{ij}||^n_n$, де n — кількість вершин в графі; l_{ij} — вага ребра між вузлами (∞ — у випадку, якщо немає ребра, 0 — якщо i = j)

Зведення матриці L в ступінь r розглядається як послідовне множення (r – 1) раз матриці на саму себе за допомогою операції Шимбелла:

$$L^k = L \Delta L^{k-1} = ||l^k_{ij}||^n_n,\ k = 2, 3, \ldots, r,$$

де Δ — операція Шимбелла, відповідно до якої, елемент l^k_{ij} визначається як мінімум з покомпонентних сум рядка *i* матриці L^k та стовпця j матриці L^{k-1} :

$$l^k_{ij} = \min[\ (l_{i1} + l^{k-1}_{1j}), (l_{i2} + l^{k-1}_{2j}), \ldots, (l_{in} + l^{k-1}_{nj})] = \min_{p=1,n}(l_{pi} + l^{n-1}_{ip}).$$

Властивості операції Шимбелла:

$$x * y = y * x \rightarrow x + y = y + x;$$
$$x + y = y + x \rightarrow \min(\ x, y\ ),$$

де l^k_{ij} — довжина найкоротшого шляху між вузлами i, j рангу k, при деякому k = k', $L^k = L^{k+1}$.

6) Метод Оттермана

Цей метод є подальшим розвитком методу Шимбелла для визначення k-найкоротших шляхів

Нехай при k = k', характеристична матриця $L^k = H = ||h_{ij}||^n_n$

1) обчислюємо H;

2) будуємо матрицю $m * L = ||f_{ij}||^n_n$ шляхом заміни головної діагоналі матриці L з 0 на ∞;

3) за допомогою операції Шимбелла визначаємо матрицю:

$$D = m * L \Delta H,\ D = ||d_{ij}||^n_n,$$

де $d_{ij} = \min[(f_{i1}+h_{1j}), (f_{i2}+h_{2j}), \ldots, (f_{ip} + h_{pj}), \ldots, (h_{in}+h_{nj}]$.

Кожен з виразів $f_{ip} + h_{pj}$ визначає довжину шляху від вузла *i* до вузла j, якщо першим проміжним вузлом після вузла *i* буде вузол p.

Величина d^1_{ij} заноситься у матрицю D^1, а значення p заноситься до матриці маршрутів M^1 під індексами *i*, j, друге мінімальне значення заноситься в D^2, p_2 в M^2.

Матричні методи дозволяють не тільки визначити величини найкоротших шляхів між усіма вершинами графу, але також одночасно отримати довжини всіх можливих шляхів між кожною парою вершин графу. Це дає можливість використовувати матричний метод також для відшукування обхідних шляхів. Обсяг обчислень при використанні матричного методу незначно залежить від структури графу.

Індивідуальні завдання

1. Побудуйте простий граф.
2. Побудуйте мультограф з петлями та дублюючими ребрами.
3. Побудуйте граф з 5-ти вершин та матрицю суміжності до нього.
4. Побудуйте орієнтований зважений граф з 5-ти вершин та матрицю суміжності до нього.
5. Побудуйте повний простий граф з 12 вершин так кістяк для нього.
6. Побудуйте ліс.
7. Завдання підвищеної складності для самостійного розв’язання на пошук циклу Ейлера — задача про прогулянку мостами Кенігсбергу (рис.1.2). На схемі позначені мости, що з'єднують острова та береги річки. Необхідно побудувати граф, визначити чи існує цикл Ейлера, і якщо він існує то визначити ребра що входять до нього.

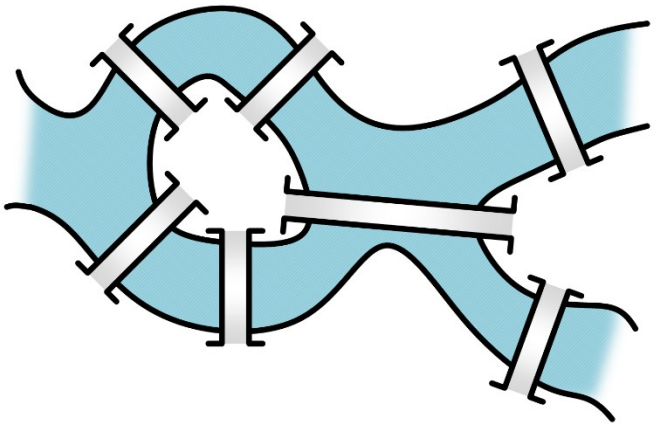

Рис.1.2 – Мости Кенігсбергу

Контрольні питання

1. В яких галузях людської діяльності застосовується теорія графів?
2. Що таке граф?
3. Наведіть приклади графів.
4. Що таке суміжні вершини графу?
5. Чим відрізняються цикл від шляху та від контуру?
6. Який тип графу має масштаб?
7. Я можна виміряти довжину шляху?
8. Що таке дерево та ліс?
9. Які властивості має кістяк?
10. Яким чином можна представити граф у ОЗП комп'ютеру?
11. Що таке матриці суміжності?
12. Як побудувати матрицю інцидентності?
13. Типи шляхів які можна найти у графі?
14. Як визначити наявність циклу Ейлера у графі?
15. Що таке мінімальне кістякове дерево?
16. Алгоритм Крускала та Прима, в чому різниця між ними?
17. Для чого застосовується топологічне сортування?
18. Що таке найкоротший шлях?
19. Які ви знаєте алгоритми пошуку найкоротшого шляху?
20. Яки умови використання методу Шимбелла для пошуку найкоротших шляхів?
21. Чим метод Оттермана кращій за метод Шимбелла?
22. Як виглядає операція Шимбелла?

2 ОСНОВИ КЛАСТЕРНОГО АНАЛІЗУ

Термін **кластерний аналіз**, було вперше введено Тріоном (Tryon) в 1939 році. Він включає в себе більше ніж 100 різноманітних алгоритмів, які відповідають на загальне питання, що виникає в різних областях людської діяльності – як організувати отримані дані в наочні структури.

Наприклад, біологи ставлять за мету розділити сукупність тварин на різні види, щоб змістовно описати відмінності між ними. Відповідно до сучасної системи, що прийнято в біології, людина належить до приматів, ссавців, амніотів, хребетних і тварин. У цій класифікації, чим вище рівень агрегації, тим менше подібності між об'єктами у відповідному класі.

В області медицини, кластеризація застосовується для класифікації симптомів захворювань та подальшого їх лікування. В археології за допомогою кластерного аналізу дослідники намагаються встановити взаємозв'язок кам'яних знарядь, об'єктів з поховань та ін. Широко застосовується кластерний аналіз у маркетингових та логістичних дослідженнях.

На відміну від завдань класифікації, кластерний аналіз не вимагає апріорних припущень про набір даних, що не накладає обмеження на подання досліджуваних об'єктів, дозволяє аналізувати показники різних типів даних (інтервальні, частоти, бінарні). При цьому необхідно пам'ятати, що змінні повинні вимірюватися в сумірних шкалах. Кластерний аналіз дозволяє скорочувати розмірність даних, робити її наочною.

Кластерний аналіз може застосовуватися до сукупностей часових рядів, з яких можна виділяти періоди схожості деяких показників і визначати групи часових рядів зі схожою динамікою.

Завдання кластерного аналізу можна об'єднати в такі групи:

1) розробка типології або класифікації;

2) дослідження корисних концептуальних схем групування об'єктів;

3) уявлення гіпотез на основі дослідження даних;

4) перевірка гіпотез або досліджень для визначення, чи дійсно типи (групи), виділені тим чи іншим способом, присутні в наявних даних.

Як правило, при практичному використанні кластерного аналізу одночасно вирішується кілька із зазначених завдань. Таким чином, кожного разу, коли необхідно класифікувати інформацію і зробити її придатною для подальшої обробки, кластерний аналіз є корисним і ефективним.

Типовим результатом кластеризації є **ієрархічне дерево**. Дерево будується починаючи з листів і "дрібних" гілок, з кожним наступним рівнем послаблюється критерій унікальності об'єктів, який визначає чи потрібно об'єднати об'єкти до одного кластеру. В результаті у кластери зв'язується все більша і більша кількість об'єктів. На останньому кроці всі об'єкти об'єднуються разом. Отримане дерево дозволяє виявити взаємозв'язок між об'єктами в наочній формі.

2.1 Визначення кластеру

Кластер визначається, як сукупність точок, що лежать на відстані не більше, ніж R від деякого "центру ваги" в m-вимірному просторі – всередині гіперсфери радіусу R або гіперкуба зі сторонами 2R.

Кластер має наступні математичні характеристики:

1) центр - це середнє геометричне місце точок у просторі змінних;

2) радіус - максимальна відстань точок від центру кластера;

3) середньоквадратичне відхилення;

4) розмір - може бути визначений або за радіусом кластера, або через середньоквадратичне відхилення об'єктів цього кластера.

Кластери можуть перекриватися, у цьому випадку неможливо за допомогою математичних процедур однозначно віднести об'єкт (або об'єкти) до одного з кластерів. Об'єкт відноситься до кластеру, якщо відстань від об'єкта до центру кластера менше радіуса кластера. Якщо ця умова виконується для двох і більше кластерів, об'єкт є спірним.

Спірний об'єкт – це об'єкт, який у міру подібності може бути віднесено до кількох кластерів.

Приклад кластеризації. Припустимо, є набір даних, що складається з 14 позицій, у кожної з яких є дві ознаки X і Y. Дані по ним наведені в таблиці 2.1.

Таблиця 2.1 – Ознаки об'єктів

№	X	Y
1	27	19
2	11	46
3	25	15
4	36	27
5	35	25
6	10	43
7	11	44
8	36	24
9	26	14
10	26	14
11	9	45
12	33	23
13	27	16
14	10	47

Для наочності зобразимо дані з таблиці на діаграмі з двома вимірами (рис.2.1). На діаграмі видно декілька груп схожих об'єктів. Об'єкти, які за сукупністю значень X та Y розташовані близько один до одного, належать до однієї групи (кластеру) – такі об’єкти будемо називати схожими; об'єкти з різних кластерів не схожі один на одного.

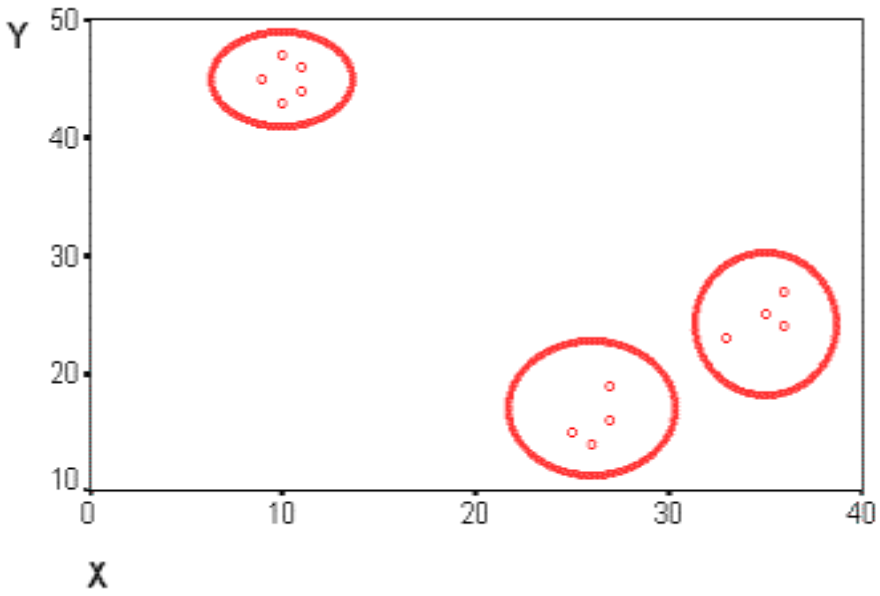

Рис.2.1 – Діаграма розсіювання змінних X і Y на площині

Для визначення схожості об’єктів необхідно виділити ознаки, за якими відбувається порівняння, і зіставити їм окремі осі координат. Таким чином схожість може бути обчислена через відстань за кожною з осей. На відстань можуть сильно впливати відмінності між осями, наприклад, якщо одна з осей вимірюється в сантиметрах, а інша в кілометрах. Ця проблема вирішується за допомогою попередньої стандартизації змінних. **Стандартизація** (standardization) або **нормування** (normalization) приводить значення всіх змінних до єдиного діапазону значень шляхом зіставлення цих значень до якоїсь величини, що відображає певні властивості конкретної ознаки.

Існують різні способи нормування вихідних даних, з яких два найпоширеніші:

1) розподіл даних за середньоквадратичним відхиленням відповідних змінних;

2) обчислення стандартизованого вкладу.

Поряд зі стандартизацією змінних, існує варіант додання кожній з осей певного **коефіцієнта важливості**, який би відображав значущість відповідної змінної. Ці коефіцієнти можуть бути отримані за допомогою експертних оцінок, що визначаються фахівцями предметної області.

Таким чином, критерієм для визначення схожості або відмінності кластерів є відстань між точками на діаграмі розсіювання. Цю схожість можна "виміряти", в наведеному прикладі вона дорівнює геометричній відстані між точками на площині. Існує декілька способів визначення **міри відстані** між кластерами або об'єктами, її називають ще **мірою близькості**.

2.2 Міри відстаней

1) **Евклідова міра** – найбільш загальна та поширена міра. Геометрична відстань в багатовимірному просторі.

$$d_e = \sqrt{\sum \left(x_i - x_j\right)^2}. \tag{4.1}$$

2) **Міра міських кварталів** (Манхеттенська міра) – сума модулю різниць координат.

$$d_m = \sum \left(\left|x_i - x_j\right|\right). \tag{4.2}$$

Для такої відстані вплив окремих викидів (великих різниць) зменшується. У більшості випадків ця міра дає такі ж результати, як і Евклідова міра.

3) **Міра Чебишева** – застосовується, якщо необхідно визначити два об'єкти як "різні", якщо вони відрізняються за будь-якою однією з координат.

$$d_c = \max\left|x_i - x_j\right|. \tag{4.3}$$

4) **Статечна міра** – застосовується, коли необхідно прогресивно збільшити або зменшити вагу певної розмірності, для якої об'єкти значно відрізняються.

$$d_p = \sqrt[1/R]{\left|x_i - x_j\right|^P}, \tag{4.4}$$

де R і P – параметри що визначаються людиною яка проводять кластеризацію. Параметр P відповідає за поступове зважування різниць за окремими координатами, а R – за прогресивне зважування великих відстаней між об'єктами.

Якщо R дорівнюватиме ½, а P дорівнюватиме двом – статечна міра буде збігатися з Евклідовою.

5) **Відсоток розбіжності** – ця міра використовується, якщо дані є категоріальними, а не числовими.

$$d_{\%} = \text{quant}\left(x_i \neq x_j\right) / N. \tag{4.5}$$

Чим більша кількість ознак у об'єктів не збігається, тим більше відстань між ними.

2.3 Правила об'єднання кластерів

На першому кроці кластеризації, коли кожен об'єкт являє собою кластер, відстані визначаються за обраною мірою. Але, в подальшому, також необхідно визначити та застосовувати правило об'єднання двох кластерів – через визначення тих точок, між якими буде вимірюватися відстань для їх об'єднання.

1) Одиночний зв'язок (метод найближчого сусіда). У цьому методі відстань між двома кластерами визначається відстанню між двома найбільш близькими об'єктами (найближчими сусідами) в різних кластерах. Отримані кластери будуть представлені довгими ланцюжками.

2) Повний зв'язок (метод найбільш віддалених сусідів). У цьому методі відстань між кластерами визначається найбільшою відстанню між будь-якими двома об'єктами в різних кластерах (тобто "найбільш віддаленими сусідами"). Цей метод зазвичай працює добре, коли кожен об’єкт належить лише одній групі і групи між собою не перетинаються.

3) Незважене попарне середнє. У цьому методі відстань між двома різними кластерами обчислюється як середня відстань між усіма об'єктами в них попарно. Метод ефективний, коли об'єкти формують відокремлені за відстанню групи, проте він працює однаково добре і для протяжних кластерів.

4) Виважене попарне середнє. Метод ідентичний попередньому, за винятком того, що при обчисленнях розмір відповідних кластерів використовується в якості вагового коефіцієнта. Цей метод використовується, коли кластери імовірно мають дуже нерівні розміри.

5) Незважений центроїдний метод. У цьому методі відстань між двома кластерами визначається як відстань між їх центрами тяжкості.

6) Зважений центроїдний метод. Цей метод ідентичний до попереднього, за винятком того, що при обчисленнях розмір відповідних кластерів використовується в якості вагового коефіцієнта. Якщо передбачаються значні відмінності у розмірах кластерів, цей метод виявляється більш доцільним за попередній.

7) Метод Варда. Цей метод відрізняється від всіх інших методів, оскільки він використовує методи дисперсійного аналізу для оцінки відстаней між кластерами. В цьому методі об'єднуються тільки ті два кластери, для яких приріст дисперсії всередині кластера є мінімальним. Метод мінімізує суму квадратів для будь-яких двох кластерів. Метод дуже ефективний, але він створює кластери малого розміру.

2.4 Методи кластеризації

Всі методи кластеризації можна розділити на дві великі групи: **ієрархічні і плоскі**. Результатом роботи ієрархічних алгоритмів є дерево ієрархічно об'єднаних кластерів, листя якого утворені обраними для кластеризації об'єктами. Плоскі алгоритми дають тільки одне єдине розбиття, без ієрархії.

Для об'єднання кластерів в ієрархічних методах використовують правила об'єднання кластерів наведені раніше (розділ 4.3). До недоліків ієрархічних алгоритмів можна віднести те, що вони будують надмірне розбиття. Зазвичай граничні нижні і верхні рівні ієрархії не використовуються.

Методи також можуть бути **багатокластерними**, коли кожному об'єкту ставиться у відповідність вектор дійсних коефіцієнтів, які показують ступінь його приналежності до кожного кластеру, і **однокластерними**, коли об'єкт належить тільки до одного кластеру (особливий випадок для багатокластерного, коли в векторі приналежності лише один коефіцієнт має значення 1, а інші – 0).

1) Побудова ієрархічного дерева знизу догори. Спочатку кожен об'єкт являє собою окремий кластер. Ці кластери поступово об'єднуються, для чого послаблюється критерій унікальності об'єктів. Іншими словами знижується поріг, який визначає об'єднувати чи ні один або більше об'єктів (кластерів) в новий загальний кластер. В результаті, у процесі побудови дерева зв'язується все більша і більша кількість об'єктів, об'єднується все більша і більша кількість кластерів. На останньому кроці всі об'єкти об'єднуються разом.

2) Побудова ієрархічного дерева зверху вниз. Спочатку всі об'єкти об'єднані в загальний кластер, а потім, з посиленням критерію унікальності об'єктів на кожному кроці, крупніший кластер розпадається на дрібніші.

3) Метод k-середніх (k-means). Цей метод відноситься до групи плоских алгоритмів. Він мінімізує середньоквадратичне відхилення розбиття за кластерами. Метод використовується, якщо вже є припущення про кількість кластерів, яку необхідно отримати. Метод дозволяє створити задану кількість кластерів що відрізняються настільки наскільки це можливо.

Алгоритм роботи методу полягає в наступному:

1) Вибрати k випадкових об'єктів, які стануть початковими центрами кластерів.

2) Для кожного з N об'єктів визначити належність до найкращого (найближчого) кластеру з k, за допомогою обраної міри відстаней.

3) Для центру кожного кластера перерахувати його положення, як середню координату об'єктів що входять до нього.

4) Повторювати п.2 до тих пір, поки не буде досягнуто критерій зупинки.

Як критерій зупинки роботи алгоритму зазвичай вибирають:

1) певну кількість ітерацій (або час роботи);

2) мінімальну зміну середньоквадратичної похибки;

3) кількість об’єктів, що перемістилися з кластера до кластеру, меншу за задану.

До недоліків алгоритму можна віднести необхідність задавати кількість кластерів для розбиття, що не завжди можливо або доцільно. Алгоритм також занадто чутливий до викидів, які можуть спотворювати середнє. Можливим вирішенням цієї проблеми є використання модифікації алгоритму - k-медіани.

4) Метод c-середніх (c-means). Являє собою модифікацію методу k-середніх, що відноситься до групи багатокластерних.

Алгоритм роботи методу полягає в наступному:

1) Вибрати початкове нечітке розбиття N об'єктів на k кластерів, для чого вибрати матрицю приналежності U розміром N × k.

2) Використовуючи матрицю U, знайти значення критерію похибки E.

3) Перегрупувати об'єкти таким чином, щоб E зменшилася.

4) Повторити п. 2, поки зміни матриці U не стануть незначними (менш заданого порогу).

Критерій похибки обчислюється за наступною формулою:

$$E=\sum^{N}_{i}\sum^{k}_{j} U_{ij}||x_i^{(k)} - c_j||^2,$$

де $c_j=\sum^{N}_{i} U_j x_i$ – центр мас кластеру j.

Цей алгоритм використовується, якщо заздалегідь відома кількість кластерів і немає необхідності однозначно віднести кожен об'єкт до одного кластеру.

2.5 Методи кластеризації засновані на графах

Суть алгоритмів кластеризації на графах полягає в тому, що вибрані для кластеризації об'єкти утворюють безліч V вершин графа G = {V, E}, та безліч ребер E, що мають вагу яка дорівнює відстані між об'єктами з V. До переваг алгоритмів цієї групи можна віднести наочність, простоту реалізації і можливість внесення різних удосконалень.

Основні алгоритми групи:

1) алгоритм виділення зв'язкових компонент;

2) алгоритм побудови мінімального кістякового дерева;

3) алгоритм пошарової кластеризації.

1) Виділення зв'язкових компонент. У графі G = {V, E} видаляються всі ребра, для яких відстані більше заданого параметра R, таким чином з'єднаними залишаються тільки найбільш близькі пари об'єктів. Суть алгоритму полягає в тому, щоб підібрати таке значення R, що лежить в діапазоні відстаней, при якому граф G природно б розділився на кілька зв'язкових компонент. Отримані компоненти і є кластерами.

Для того щоб обрати параметр R будується гістограма розподілів попарних відстаней. У даних з добре вираженою кластерною структурою, на цій гістограмі буде два піки:

1) той, що відповідає серединно-кластерним відстаням;

2) той, що відповідає між-кластерним відстаням.

Параметр R вибирається як один з мінімумів між цими піками.

До недоліку алгоритму можна віднести те що, управляти кількістю кластерів за допомогою такого порогу досить важко.

2) Алгоритми мінімального кістякового дерева. Спочатку необхідно визначити мінімальне кістякове дерево для G. Це таке дерево в якому сума ваг ребер

мінімальна. Зазвичай воно визначається за допомогою алгоритмів Крускала (Kruskal) або Прима.

Алгоритм Крускала полягає в сортуванні всіх ребер в порядку зростання довжини, і додаванню їх по черзі в мінімальний кістяк, якщо вони з'єднують різні компоненти зв'язності.

Алгоритм Прима. Побудова починається з дерева, що включає в себе одну (довільну) вершину. Протягом роботи алгоритму дерево розростається, поки не охопить всі вершини вихідного графа. На кожному кроці алгоритму до дерева приєднується найлегше з тих ребер, що з'єднують вершину з побудованого дерева і вершину, що не належить дереву.

Після визначення кістякового дерева, з нього, послідовно видаляються ребра з найбільшою вагою до тих пір поки не буде отримано необхідну кількість кластерів.

3) Алгоритм пошарової кластеризації засновано на виділенні зв'язкових компонент графа на деякому рівні відстаней між об'єктами (вершинами). Рівень відстані задається порогом відстані R. Цей алгоритм формує послідовність підграфів для графа G, які відображають ієрархічні зв'язки між кластерами. Одночасно він може створювати як плоске розбиття даних так і ієрархічне. У таблиці 2.2 наведено обчислювальну складність методів кластеризації.

Таблиця 2.2 - Обчислювальна складність методів кластеризації

Метод	Обчислювальна складність
Ієрархічний (побудова дерева)	$O(N^2)$
k-середніх	$O(N \cdot k \cdot i)$, де k – число кластерів, i – число ітерацій
c-середніх	
Виділення зв'язкових компонент	залежить від алгоритму
Мінімальне кістякове дерево	$O(N^2 \cdot \log N)$
Пошарова кластеризація	$O(\max(N, M))$, де $M < N \cdot (N-1)/2$

Індивідуальні завдання

Провести кластерізацію за методом к-середніх, дослідити процес кластерізації

№	міра для дослідження	розмір групи (mnPts, mxPts)	№	міра для дослідження	розмір групи (mnPts, mxPts)
1	Міських кварталів	10, 80	11	Чебишева	24, 48
2	Евклідова	16, 64	12	Міських кварталів	32, 48
3	Чебишева	20, 100	13	Чебишева	50, 60
4	Міських кварталів	15, 30	14	Міських кварталів	25, 89
5	Чебишева	22, 56	15	Евклідова	16, 64
6	Міських кварталів	32, 64	16	Міських кварталів	32, 80
7	Евклідова	100, 110	17	Евклідова	20, 50
8	Міських кварталів	55, 80	18	Чебишева	20, 40
9	Чебишева	64, 128	19	Міських кварталів	60, 70
10	Евклідова	18, 29	20	Евклідова	16, 64

Кількість кластерів numG отримати за формулою:
numG = (номер варіанту * 97) % 25 + 10;
Дисперсію групи maxRad отримати за формулою:
maxRad = ((номер варіанту * 491) % 50) / 200 + 0,05;

Контрольні питання

1. Що таке кластерний аналіз?
2. Які завдання кластерного аналізу?
3. Що є типовим результатом кластерізації?
4. Як визначається кластер?
5. Що таке спірний об'єкт?
6. Що таке міри відстані?
7. Для чого застосовуються міри відстані?
8. В чому різниця між Евклідовою та Манхеттенською мірами?
9. В якому випадку доцільно використовувати відсоток розбіжності?
10. Яки правила об'єднання кластерів ви знаєте?
11. В чому різниці між методом одиночного з'вязку та методом попарного середнього?
12. Яки групи методів кластерізації вам відомі?
13. Яки особливості методу к-середніх?
14. Що можна сказати про складність різних методів кластерізації?

3 АЛГОРИТМИ СОРТУВАННЯ

Алгоритми сортування мають велике практичне застосування, їх можна зустріти майже всюди, де мова йде про обробку та зберігання великих обсягів інформації. Деякі завдання обробки даних вирішуються простіше, якщо дані впорядковані.

Сортування застосовується у всіх без винятку областях програмування, наприклад, базах даних або математичних програмах. Від ефективності, і перш за все швидкості їх виконання багато в чому залежить ефективність роботи всієї програми.

Сортування – упорядкування елементів послідовності за певним законом. Весь процес сортування складається з **кроків**, на кожному з яких обробляється один (або пара) елемент послідовності. Кроки об'єднуються в **ітерацію** – одноразову обробку всіх елементів послідовності (або її частини) і, в підсумку, кілька ітерацій складають повне сортування.

Всі методи сортування можна умовно розділити на три великі групи: елементарні, комплексні та спеціальні.

1) Елементарні методи засновані на впорядкуванні одного елемента послідовності за одну ітерацію, при цьому за кожен крок сортування елемент може бути переставлено тільки на одну позицію. Елементарні методи просто реалізувати і вони не потребують додаткової пам'яті.

2) Комплексні методи засновані на впорядкуванні кількох елементів за одну ітерацію, при цьому кожен елемент може бути переставлено на відстань більше ніж 1. Методи цієї групи відносно складні в реалізації і вимагають додаткової пам'яті для своєї роботи.

3) Спеціальні методи засновані на обмеженні типів або діапазонів елементів що впорядковуються, використанні спеціального обладнання, або значних витратах пам'яті.

Типова складність для методів сортування:

- елементарні $O(N^2)$;
- комплексні $O(N \cdot \log N)$ або $O(m \cdot N)$, де $m << N$;
- спеціальні $O(m \cdot N)$, де $m << N$ або $O(N)$.

Таким чином ми бачимо, що сортування не може мати складність гірше за $O(N^2)$, або краще за $O(N)$.

При оцінці складності для сортувань розглядається окремо кількість проведених обмінів (переміщень) та порівнянь.

При виборі алгоритмів сортування необхідно поставити перед собою питання: чи існує найкращий алгоритм? Маючи приблизні характеристики вхідних даних, можна підібрати метод, який працює найбільш оптимальним чином.

Методи сортування можна розділити на **стійкі та нестійкі**.

Стійкістю називається така властивість методу сортування, при якому, після його роботи зберігається порядок розміщення елементів в послідовності, яка містить елементи з однаковими ключами.

Якщо кількість кроків необхідних для сортування послідовності залежить від значень їх ключів або взаємного розташування, такий алгоритм називається

адаптивним. Якщо кількість кроків залежить тільки від кількості елементів в послідовності що впорядковується, такий алгоритм є **неадаптивним.**

Сортування буває **зовнішнім та внутрішнім**. Якщо всі елементи послідовності (і додаткові дані) можна розмістити в ОЗП обчислювальної системи, таке сортування називається внутрішнім. Якщо частина елементів розміщується на зовнішніх (повільних) носіях, таке сортування називається зовнішнім. Для внутрішніх сортувань прагнуть скоротити число порівнянь та інших операцій, а для зовнішніх сортувань, крім цього вирішальним фактором є кількість операцій по введенню і виведенню даних, тому що доступ до даних на зовнішньому носії здійснюється набагато повільніше, ніж операції з ОЗП.

Сортування буває **пряме та непряме.** Якщо елементи послідовності мають великий розмір, то доцільно виконувати сортування їх індексів (номерів), а не самих елементів. Після чого необхідно перемістити елементи згідно з новим розташуванням індексів. Таке сортування називається **непрямим**.

Послідовність у всіх прикладах представлена у вигляді масиву A[N].

В подальшому будуть розглянуті тільки методи внутрішнього прямого сортування. Сортування буде проводитись за зростанням, з використанням оператора "<". У якості елементів використовуватимуться цілі числа, для яких ключ та дані збігаються, тому питання стійкості також не буде розглянуто.

Для обміну елементів буде використовуватися функція:

```
void swap( int& a, int& b )
{
        int c = a;
        a = b;
        b = c;
}
```

3.1 Елементарні методи

1) Сортування простим обміном. Інша назва методу – бульбашкове сортування. Метод заснований на перегляді елементів послідовності з обміном місцями сусідніх елементів що порушують заданий порядок. Ітерації проводяться до тих пір, поки послідовність не буде остаточно відсортована (не було жодного обміну за останню ітерацію).

Складність в середньому і найгіршому випадках для обмінів і порівнянь складає $O(N^2 / 2)$.

```
void sort_simple_swap( int* A, int N )
{
  bool exch;
  do
  {
```

```
            exch = false;
            --N;
            for( int i = 0; i < N; ++i )
            {
                    if( A[i + 1] < A[i] )
                    {
                            swap( A[i], A[i + 1] );
                            exch = true;
                    }

            }// for( int i = 0; i < N; ++i )
    }
    while( exch );

}
```

Цей алгоритм може бути поліпшено, якщо запам'ятовувати не тільки сам факт обміну на ітерації, але і індекс елементу з останнього обміну j. Всі пари сусідніх елементів з індексами, більшими за j, вже розташовані в потрібному порядку. Подальші ітерації можна закінчувати на індексі j, замість того щоб рухатися до встановленої заздалегідь верхньої межі.

```
void sort_simple_swap_opt( int* A, int N )
{
    int j = N - 1;
    do
    {
            int k = -1;
            for( int i = 0; i < j; ++i )
            {
                    if( A[i] > A[i + 1] )
                    {
                            swap( A[i], A[i + 1] );
                            k = i;
                    }

            }// for( int i = 0; i < j; ++i )
            j = k;
    }
    while( j >= 0 );

}
```

Ще одне поліпшення алгоритму можна отримати з наступного спостереження. Велика бульбашка (елемент з великим значенням) знизу підніметься наверх за одну

ітерацію, але менші бульбашки (елементи з малими значеннями) опускаються з мінімальною швидкістю: один крок за ітерацію. Так що послідовність {6, 1, 2, 3, 4, 5} буде впорядковано за 1 прохід, а сортування послідовності {2, 3, 4, 5, 6, 1} потребує 5 ітерацій.

Щоб уникнути подібного ефекту, треба змінювати напрямок проведення наступних ітерацій. Одержаний алгоритм називають "шейкер-сортування".

```
void sort_shaker( int* A, int N )
{
  int j, l = 0, r = N - 1;
  do
  {
          j = -1;
          for( int i = l; i < r; ++i )
          {
                  if( A[i] > A[i + 1] )
                  {
                          swap( A[i], A[i + 1] );
                  j = i;
                  }
          }// for( int i = l; i < r; ++i )

          if( j < 0 ) break;
          r = j;
          j = -1;
          for( int i = r; i > l; --i )
          {
                  if( A[i] < A[i - 1] )
                  {
                          swap( A[i], A[i - 1] );
                          j = i;
                  }
          }// for( int i = r; i > l; --i )

          if( j < 0 ) break;
          l = j;
  }
  while( l < r );
}
```

Недоліки:

● найповільніший алгоритм сортування з існуючих.

Переваги:

● не вимагає додаткової пам'яті;

● добре працює при сортуванні частково або практично повністю впорядкованої послідовності;

● простота реалізації;

● крок сортування виконується дуже швидко.

2) Сортування простим вибором. Відшукується найменший елемент послідовності, потім він міняється місцями з першим елементом послідовності. Далі, в решті послідовності (без 1-го елемента), знаходиться другий мінімальний елемент і міняється місцями з другим елементом послідовності, цей процес повторюється N-1 раз, на кожній ітерації послідовність для сортування скорочується на 1 елемент. Для знаходження найменшого елемента алгоритм робить N-1 порівнянь.

В середньому і найгіршому випадках кількість обмінів O(N), кількість порівнянь O(N^2 / 2).

```
int find_min_index( const int* A, int N )
{
  int k = 0;
  for( int j = k + 1; j < N; ++j )
  {
          if( A[j] < A[k] ) k = j;
  }
  return k;

}

void sort_simple_select( int* A, int N )
{
  for( int i = 0; i < N - 1; ++i )
  {
          swap( A[i], A[ find_min_index( A + i, N - i ) ] );
  }

}
```

Недоліки:

● кількість операцій не залежить від того наскільки вже впорядкована послідовність.

Переваги:

● не вимагає додаткової пам'яті;

● завжди постійна кількість обмінів;

● може використовуватися як альтернатива непрямому сортуванню, якщо ключі мають незначний розмір, а дані – великий, тобто витрати на переміщення даних істотно більше, ніж витрати на порівняння ключів.

3) Сортування простими вставками. Метод сортування застосовується в карткових іграх – кожен новий елемент вставляється в належне місце серед вже впорядкованих, при цьому елементи, що знаходяться праворуч від місця вставки зсуваються на одну позицію вправо. В даному методі не потрібна операція повноцінного обміну для двох елементів.

В середньому кількість полуобмінів (переміщень) і порівнянь складає O(N^2 / 4). У гіршому випадку в два рази більше переміщень, O(N^2 / 2).

```
void insert_to( int* A, int i, int e )
{
  int j = i;
  for( ; j > 0; --j )
  {
          if( A[j - 1] < e ) break;
          A[j] = A[j - 1];
  }
  A[j] = e;
}

void sort_simple_insert( int* A, int N )
{
  for( int i = 1; i < N; ++i )
  {
          insert_to( A, i, A[i] );
  }
}
```

Недоліки:

● складність сортування сильно залежить від порядку ключів у вхідній послідовності.

Переваги:

● не вимагає додаткової пам'яті;

● відмінно працює при сортуванні частково впорядкованої послідовності;

● немає необхідності знати всі елементи до початку (і в процесі) сортування, можна сортувати по мірі їх надходження.

Даний метод зазвичай використовується як сумісний з одним з комплексних методів сортування, для досортування коротких послідовностей.

3.2 Комплексні методи сортування

1) Швидке сортування або алгоритм Хоара. Алгоритм функціонує за принципом «розділяй і володарюй». Теоретично це найшвидший метод сортування загального призначення, який не накладає обмежень на ключі.

На кожній ітерації алгоритм розділяє послідовність на дві частини, а потім сортує ці частини незалежно одну від одної. У процесі сортування відбувається переупорядкування послідовності таким чином, що виконуються наступні умови:

- A[i] для деякого i займає свою остаточну позицію в послідовності;
- жоден з елементів A[1],…,A[i-1] не перевищує A[i];
- жоден з елементів A[i + 1],…, A[N] не менше за A[i].

Повне сортування досягається розподілом послідовності на підпослідовності, для яких виконуються зазначені умови, з подальшим застосуванням отриманого алгоритму к ним.

На практиці, перш за все, вибирається елемент A[i], що розділяє послідовність. Далі відбувається перегляд послідовності зліва до тих пір, поки не буде знайдено елемент з ключем, для якого виконується нерівність A[l].k > A[i].k. Потім переглядається послідовність справа, поки не буде знайдено елемент з ключем, для якого виконується нерівність A[r].k < A[i].k. Коли обидва елемента знайдено, відбувається обмін A[l] та A[r]. Таким чином, за один крок елементи послідовності можуть пересуватися на будь яку відстань, на противагу з елементарними методами сортування. Далі процес повторюється вже з позицій i + 1 та r - 1 для лівої і правої частин відповідно.

Якщо зліва або справа не знайдено елемент A[i] або A[r] з ключем, що відповідає умовам, то для обміну використовується A[i] з відповідною зміною значення i на l або r. Ітерація припиняється коли l = i = r - вся послідовність відповідає умовам.

Далі незалежно упорядковуються частини розділені A[i], а елемент A[i] займає своє остаточне положення у впорядкованій послідовності і в сортуванні більш не приймає участі.

Швидке сортування в найгіршому випадку виконує приблизно $N^2/2$ порівнянь, а в середньому випадку - приблизно 2N(log N) порівнянь. Якщо менша з двох підпослідовностей сортується першою, то витрати пам'яті в середньому не перевищують log N.

Важливою частиною, від якої залежить ефективність алгоритму, є **початковий вибір елементу** A[i], що розділяє послідовність. Цей вибір може здійснюватися наступним чином:

- перший, або останній елемент (i = 0 або i = N - 1);
- середній елемент (i = N/2);
- випадковий елемент;
- медіана з трьох елементів (початок, кінець, середина).

Останні два варіанти є найкращими і дають мінімальну ймовірність виникнення найгіршого випадку, який виникає при зворотній впорядкованості елементів послідовності.

Для підпослідовностей невеликого розміру доцільно застосовувати простий метод сортування, наприклад, сортування простими вставками. Саме за таким принципом працює стандартний алгоритм сортування мови C++ (std::sort).

3) Сортування за розрядами. Як випливає з назви, в даному випадку при сортуванні розглядається не повне значення ключа елементу, а його розряди. Ключ кожного елемента повинен бути представлений у вигляді числа.

Всі алгоритми сортування за розрядами можна розділити на 2 групи: MSD (Most Significant Digit) - сортування за старшим розрядом, LSD (Least Significant Digit) - сортування за молодшим розрядои.

Швидке двійкове сортування. Відноситься до групи MSD. Алгоритм схожий на швидке сортування. Ключі розглядаються в двійковій системі. На кожній ітерації необхідно згуртувати усі елементи з ключами зі значенням 0 у старшому розряді у лівій частині масиву, а зі значенням 1 - в правій. Для двох отриманих підпослідовностей необхідно повторити процес впорядкування за наступним у бік зменшення розрядом ключів, і так далі, поки не буде досягнуто молодшого розряду числа.

З практичної точки зору, необхідно зліва знайти ключ зі значенням 1 у відповідному розряді, а справа - зі значенням 0, та обміняти ці елементи місцями. Таким чином повторювати до тої пори, поки перегляди зліва і справа не зустрінуться. Місце де цифра 0 змінюється на 1 і є місцем розділення на підмножини для подальшого сортування. Для цього методу сортування немає елементу, що розділює послідовність, як у швидкому сортуванні.

Швидкодія методу залежить від кількості m значущих розрядів числа і пропорційна mN.

Сортування за молодшим розрядом (група LSD) - необхідно розкласти усі числа послідовності до d черг (d - основа системи числення) за молодшим розрядом ключа. На другому етапі, необхідно відновити початкову послідовність, послідовно отримуючи елементи з черги. Ця послідовність буде використовуватись на наступній ітерації.

Витрати пам'яті - N елементів для зберігання черг.

Складність алгоритму O(mN), де m - кількість значущих розрядів в ключі.

Важливою частиною алгоритмів цього типу є функція виділення i-го розряду з ключа k. Цю функцію в загальному випадку можна визначити як

$$d_i = (k/d^i) \bmod d.$$

Для ефективної роботи алгоритмів доцільно вибирати основу системи числення, що дорівнює ступеню двійки $d = 2^p$. Тоді операцію піднесення до ступеню можна замінити операцією побітового зсуву, а операцію взяття остачі – операцією побітового множення. Функція прийме наступний вигляд

di = (k >> [i p]) & (d - 1).

Для двійкової системи функцію можна ще скоротити до виду

$d_i = (k >> i) \,\&\, 1.$

3.3 Спеціальні сортування

1) Сортування методом підрахунку (гістограми). Цей алгоритм сортування заснований на використанні специфічних властивостей (обмежень) ключів. Ключі використовуються у якості індексів в масиві гістограм і повинні бути цілими числами. Таким чином, цей алгоритм можливо застосовувати тільки у випадку, якщо ключі та дані у елементі співпадають, наприклад, у методах обробки зображень.

Сортування складається з двох етапів:

1) Побудова гістограми - підрахунок кількості входжень у послідовність для кожного ключа. Кожен елемент гістограми має номер відповідного ключа, та містить лічильник входжень ключа до послідовності.

2) Відновлення впорядкованого масиву за допомогою гістограми. Гістограма опрацьовується з початку (або з кінця, якщо потребується упорядкування від більшого до меншого), кожен елемент записується до вихідної упорядкованої послідовності стільки разів, скільки є у відповідному лічильнику.

Основна вимога для ефективного функціонування алгоритму - ключі мають бути розподілені в невеликому діапазоні.

Складність алгоритму - O(2N). Також необхідна додаткова пам'ять для зберігання масиву лічильників (гістограми).

2) Метод Бетчера. Або парно-непарне сортування злиттям. В основі методу лежить операція порівняння-обміну та алгоритми тасування (shuffle) і зворотного тасування (unshuffle). За допомогою операції unshuffle ми розбиваємо послідовність на дві половини, які впорядковуються незалежно, а після цього зливаються назад за допомогою операції shuffle.

Операція shuffle ділить послідовність навпіл і далі перший елемент першої половини - перший на виході, а перший елемент другої половини - другий на виході, другий елемент першої половини - третій на виході і т. д. Послідовність обов'язково повинна бути парної довжини. Фактично, операція розставляє елементи з першої половини по парних позиціях, а з другої - по непарних.

Операція unshuffle зворотна до попередньої. Елементи, що займають парні позиції, відправляються в першу половину виходу, елементи з непарних позицій - в другу.

Алгоритм розділяє послідовність на дві частини, які сортуються окремо одна від одної. Після чого послідовності зливаються в одну загальну послідовність. Фактично сортування алгоритмом Бетчера є різновидом сортування злиттям і діє як 2^p сортування, 2^{p-1} сортування, …, 8 сортування, 4 сортування, 2 сортування і 1 сортування, але в рамках ітерації елементи послідовності не перекриваються. Таким чином, в алгоритмі Бетчера відбувається злиття пар відсортованих послідовностей.

Цей алгоритм є прикладом неадаптивного алгоритму в якому послідовність операцій залежить тільки від кількості елементів, але не залежить від послідовності вхідних даних.

Для роботи з послідовностями, розмір яких не є ступеню двійки застосовується два способи. Найпростіший спосіб - додати необхідну кількість елементів, щоб розмір послідовності став дорівнювати ступеню двійки. Значення доданих елементів при

цьому мають бути апріорі більшими (або меншими) за будь-який елемент у початковій послідовності.

Більш ефективний підхід полягає в наступному. Оскільки будь-яке число можна представити як суму ступенів двійки, то можливо розбити початкову послідовність на такі підпослідовності, в яких кількість елементів відповідає таким ступеням. Кожну з послідовностей впорядкувати окремо сортуванням Бетчера і об'єднати їх за допомогою операції злиття. При цьому маленькі підпослідовності можливо сортувати іншим алгоритмом, який, наприклад, не вимагає рекурсивних викликів.

Алгоритм сортування Бетчера не є ефективним алгоритмом сортування, проте він має одну важливу рису: всі порівняння та обміни на одній ітерації можна виконувати одночасно. Наприклад, 1024 елемента можна розсортувати методом Бетчера всього за 55 паралельних кроків. Таким чином, цей метод сортування може виконуватися паралельно при наявності декількох виконавчих блоків процесору.

Сортування виконується за 1/2 [log N] ([log N] +1) паралельних кроків.

3) Сортувальні сітки це абстрактні обчислювальні машини, які здатні здійснювати доступ до елементів тільки за допомогою операції порівняння та обміну. Вони складаються з автономних модулів порівняння-обміну (компараторів).

Сортувальні сітки прийнято зображати у такий спосіб, де сортовані елементи послідовності позначаються горизонтальними лініями даних, а компаратори - вертикальними відрізками, які з'єднують тільки дві лінії (рис.3.1).

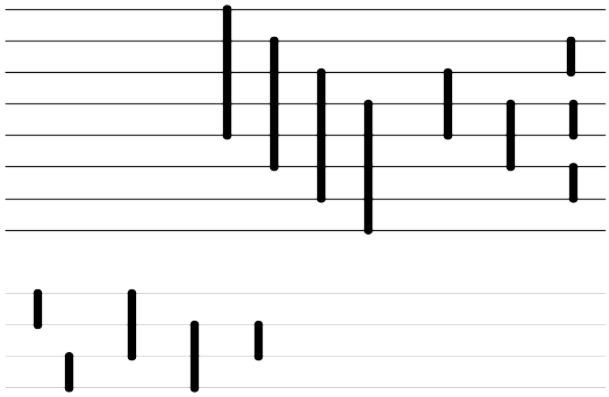

Рис.3.1 - Приклади сортувальних сіток

На даний момент спосіб побудови сіток, що забезпечують мінімальний час сортування для довільного числа входів, невідомий. Сітки можуть бути побудовані із неадаптивних алгоритмів сортування, як списки пар номерів елементів які потрібно порівнювати та обміняти.

Наприклад, сітки за методом Бетчера використовують $N(\log N)^2/4$ компараторів, сортування у яких може бути виконане за $(\log N)^2/2$ паралельних кроків.

В контейнер будуть записані пари, які складають компаратори сортувальної сітки Бетчеру. Треба пам'ятати що, сітка Бетчера може бути побудована тільки для розмірів послідовності size2, що є ступенями двійки.

Сортувальні сітки доцільно використовувати якщо сортування необхідно проводити апаратно, а не програмно.

Алгоритм отримання сортувальної сітки (контейнер net) за допомогою методу Бетчера:

```
void betcher_sort_net( vector<int>& net, int size2 )
{

    for( int p = size2; p > 0; p /= 2 )
    {
        for( int q = size2, r = 0, d = p; ; d = q - p, q /= 2, r = p )
        {
            for( int i = 0; i < size2 - d; ++i )
            {
                if( ( i & p ) == r )
                {
                    net.push_back( i );
                    net.push_back( i + d );

                }// if( ( i & p ) == r )

            }// for( int i = 0; i < size2 - d; ++i )

            if( q == p ) break;

        }// for( int q = size2, r = 0, d = p; ; d = q - p, q /= 2, r = p )

    }// for( int p = size2; p > 0; p /= 2 )
}
```

3.4 Знаходження медіани числової послідовності

Крім упорядкування послідовності, на практиці зустрічається завдання пошуку медіани послідовності. **Медіана** - елемент послідовності M, для якого виконується правило: половина елементів послідовності менша або дорівнює M, а половина більша, або дорівнює M.

Не слід плутати середнє арифметичне і медіану. Найпростішим способом знайти медіану є сортування послідовності і взяття елемента, що розташований по середині. Таким чином, можна визначити медіану ще й як елемент, що знаходиться посередині відсортованої послідовності. Вочевидь, для пошуку медіани можна пристосувати який-небудь із методів сортування.

Основні методи визначення медіани:

- сортування послідовності і вибір середнього елемента;
- модифікований метод підрахунку;
- модифіковане швидке сортування;

- оптимізована сітка Бетчера для пошуку медіани;
- спеціалізована сітка для пошуку медіани.

Для пошуку медіани немає необхідності повністю сортувати послідовність, досить знайти елемент який зайняв свою позицію посередині впорядкованої послідовності, після чого сортування можна припиняти.

У методі підрахунку, на другому етапі немає необхідності відновлювати впорядковану послідовність, досить підсумувати лічильники гістограми, до тих пір, доки сума не досягне або перевищить значення N/2. Той елемент на якому це перевищення відбулося і буде медіаною.

У методі швидкого сортування необхідно додати дві додаткові умови:

1) Якщо після ітерації остаточна позиція елемента, що розділяє послідовність, дорівнює N/2, то медіану вже знайдено - це буде елемент, що розділяє послідовність (це випливає з властивостей сортування).

2) Немає необхідності сортувати ту частину послідовності в яку не входить елемент з номером, що дорівнює N/2.

Таким чином, в середньому медіана може бути знайдена зі складністю O(N).

Сортувальну сітку, що побудована за методом Бетчера, необхідно оптимізувати: прибрати компаратори, які не впливають на центральний елемент. Також для пошуку медіани застосовуються спеціалізовані сортувальні сітки. Відомі сітки, близькі до теоретично оптимальних, для пошуку медіани з послідовності до 100 елементів.

Індивідуальні завдання

Провести сортування простим методом, та дослідити роботу алгоритму сортування

№	алгоритм для дослідження	розмір (N), тип та діапазон вихідних даних	№	алгоритм для дослідження	розмір (N), тип та діапазон вихідних даних
1	Сортування простим вибором	3000, float, [-3e3, 1e5]	11	Сортування простими вставками	2100, int, [-1100, 2000]
2	Сортування простим обміном	2000, short, [-10, 20]	12	Сортування простим обміном	2300, float, [-1e3, 2e4]
3	Сортування простими вставками	3200, char, [-100, 100]	13	Сортування простим вибором	8000, unsigned int, [1000, 10000]
4	Шейкер-сортування	5100, long, [-1000, 12000]	14	Сортування простими вставками	2000, double, [-0.11, 0.325]
5	Сортування простим вибором	1500, unsigned short, [8000, 31000]	15	Сортування простим обміном	2000, unsigned int, [0, 10000]
6	Сортування простим обміном	5660, double, [-5e5, 0]	16	Шейкер-сортування	4600, short, [-300, 500]
7	Сортування простим вибором	3650, unsigned char, ['A', 'Z']	17	Сортування простим вибором	7000, double, [-1, 1]
8	Шейкер-сортування	3500, long, [-5, 5350]	18	Сортування простим обміном	8400, short, [-16000, 1]
9	Сортування простими вставками	2500, short, [-30000, 30000]	19	Шейкер-сортування	5000, float, [-10, 10]
10	Сортування простим обміном	4200, char, ['0', '9']	20	Сортування простими вставками	11000, long, [-100, 50000]

Провести сортування комплексним методом, та дослідити роботу алгоритму сортування

	№	алгоритм для дослідження	розмір (N), тип та діапазон вихідних даних		№	алгоритм для дослідження	розмір (N), тип та діапазон вихідних даних
	1	ШДС	2500, int, [100, 1500]	**1**	**1**	ШС з вибором місця розподілу за допомогою ГПВЧ	2300, float, [-1e3, 2e4]
	2	СЗМР, 10 система числення	2000, short, [30, 1020]	**2**	**1**	СЗМР, 16 система числення	2100, int, [200, 20200]
	3	ШДС	2980, unsigned char, [20, 80]	**3**	**1**	ШДС	4675, unsigned int, [1000, 10000]
	4	ШС з вибором місця розподілу за допомогою ГПВЧ	5100, long, [-1000, 12000]	**4**	**1**	СЗМР, 8 система числення	3250, char, ['0', '9']
	5	СЗМР, 4 система числення	4200, char, ['0', 'Z']	**5**	**1**	ШДС	2200, unsigned int, [0, 10000]
	6	ШДС	5900, short, [0, 500]	**6**	**1**	ШС з вибором місця розподілу по середині послідовності	5660, double, [-5e5, 0]
	7	ШС з вибором місця розподілу по середині послідовності	2500, short, [-30000, 30000]	**7**	**1**	ШС з вибором місця розподілу за допомогою медіани	2000, double, [-0.11, 0.325]
	8	СЗМР, 10 система числення	3500, long, [500, 5350]	**8**	**1**	ШДС	6400, unsigned short, [16, 65000]
	9	ШС з вибором місця розподілу по середині послідовності	4200, double, [-1.3, 1.457]	**9**	**1**	ШС з вибором місця розподілу за допомогою ГПВЧ	5000, float, [-10, 10]
0	**1**	СЗМР, 2 система числення	1500, short, [5000, 23000]	**0**	**2**	ШС з вибором місця розподілу за допомогою медіани	5300, float, [-1e5, 3e3]

ШС - швидке сортування,

ШДС - швидке двійкове сортування,

СЗМР - сортування за молодшим розрядом.

Провести сортування спеціальним методом, та дослідити роботу алгоритму сортування

№	алгоритм для дослідження	розмір (N), тип та діапазон вихідних даних	№	алгоритм для дослідження	розмір (N), тип та діапазон вихідних даних
1	Швидке сортування	18000, float, [-1e3, 4e4]	**11**	Швидке сортування	11300, int, [-1000, 200]
2	Метод Бетчера	8192, unsigned long, [10, 50000]	**12**	Метод підрахунку	15000, short, [100, 2000]
3	Метод підрахунку	10000, char, [0, 100]	**13**	Швидке сортування	9900, unsigned int, [1000, 5000]
4	Швидке сортування	8500, unsigned short, ['0', '9']	**14**	Метод Бетчера	8192, float, [1.f, 12.f]
5	Метод підрахунку	7500, unsigned char, ['A', 'Z']	**15**	Швидке сортування	7650, long, [-1600, 90]
6	Швидке сортування	15015, short, [-16384, 16384]	**16**	Метод Бетчера	4096, char, [-100, 100]
7	Метод Бетчера	16384, unsigned int, [100, 500]	**17**	Метод підрахунку	11000, unsigned short, [500, 1000]
8	Швидке сортування	10500, long, [-30000, 50000]	**18**	Швидке сортування	5555, double, [-1e7, 1e3]
9	Метод Бетчера	4096, double, [-10.0, 10.0]	**19**	Метод підрахунку	15000, char, [32, 'a']
10	Метод підрахунку	13500, char, ['0', 'X']	**20**	Метод Бетчера	16384, long, [0, 570]

Контрольні питання

1. Що таке сортування?
2. Чим відрізняються елементарні методи сортування від комплексних?
3. Які методи сортування можливо віднести до групі спеціальних?
4. Яка типова складність методів сортування з різних груп?
5. Що таке стійкість методу сортування?
6. Які методи сортування являються адаптивними?
7. Що таке зовнішній метод сортування?
8. Яке сортування називається непрямим?
9. Яки недоліки методу сортування простим обміном?
10. Яки переваги у методу сортуванні простим вибором?
11. Від чого залежить складність сортування простими вставками?
12. За яким принципом функціонує алгоритм Хоара (швидке сортування)?
13. Як доцільно вибирати елемент що розділяє в методі Хоару?
14. Які ви знаєте особливості практичної реалізації методу швидкого сортування?
15. Які обмеження для застосування методів сортування за розрядами?
16. Яка складність алгоритму швидкого двійкового сортування?
17. Як виділити і-й розряд з ключа?
18. В чому перевага застосування двійкової системи обчислення для алгоритмів сортування за розрядами?
19. В чому особливості та переваги сортування методом підрахунку?
20. В яких умовах алгоритм Бетчера має переваги перед іншими методами сортування?
21. Як застосувати метод Бетчера для сортування послідовностей розмір яких не є ступенем двійки?
22. Що таке сортувальна сітка?
23. Як можна отримати сортувальну сітку?
24. Що таке медіана послідовності?
25. Як треба модифікувати алгоритм швидкого сортування для того щоб він став методом пошуку медіани зі складністю O(N)?
26. Як треба модифікувати алгоритм підрахунку для того щоб він став методом пошуку медіани числової послідовності?

Список рекомендованої літератури

1. Stroustrup, Bjarne. The C++ programming language / Bjarne Stroustrup.— Fourth edition. Published by Pearson Education, Inc. 2013. 1360 p. ISBN 978-0-321-56384-2.
2. Levitin A. Introduction to the design & analysis of algorithms / Anany Levitin. 3rd ed. Pearson Education, Inc., publishing as Addison-Wesley. 2012. 593 p. ISBN 10: 0-13-231681-1.
3. Алгоритми і структури даних: лабораторний практикум для студентів напряму підготовки "Телекомунікації і радіотехніка" / Уклад.: М.А. Скулиш, С.В. Суліма,. – К.: КПІ ім. Ігоря Сікорського, 2021. –109с.
4. Alsuwaiyel, M. H. Algorithms Design Techniques and Analysis/ M. H. Alsuwaiyel. World Scientific Publishing Co. Pte. Ltd., New Jersey : World Scientific, 2016. 571 p. ISBN 9789814723640.
5. Sedgewick R. and Wayne K. Algorithms 4th Edition. Robert Sedgewick and Kevin Wayne. Pearson Education, Inc. 2011. 955 p. ISBN-13: 978-0-321-57351-3.
6. Maarten van Steen. An Introduction to Graph Theory and Complex Networks. Maarten van Steen. 2010. 300 p. ISBN-13: 978-9081540612.
7. Graph Theory - Advanced Algorithms and Applications. Edited by Beril Sirmacek. Published in Croatia, 2018 by INTECH d.o.o. 2018. 198 p. ISBN 978-953-51-3772-6.
8. Sedgewick R. Algorithms Third Edition in C++. Part 5 Graph Algorithms/ Robert Sedgewick.—3d ed. Princeton University. Published by Pearson Education 2006. 670 p. ISBN 0-201-36118-3.
9. Pang.N Ing Tan, Steinbach M., Vi Pi N Ku Mar. Introduction to Data Mining (Second Edition). Pearson Education, Inc. 2006. 791 p. ISBN 0-321-42052.
10. Data clustering : algorithms and applications / [edited by] Charu C. Aggarwal, Chandan K. Reddy. Data Mining and Knowledge Discovery Series. University of Minnesota. CRC Press. 2014. 622 p. ISBN 978-1-4665-5821-2.

ЗМІСТ

Навчальне видання

ЗУЄВ Андрій Олександрович
КАРАМАН Дмитро Григорович
ГАПОН Дмитро Анатолійович

МЕТОДИ УПОРЯДКУВАННЯ ДАНИХ

Методичні вказівки
до виконання самостійних робіт

з курсу «Інформаційні технології та програмування»

для студентів спеціальностей «Автоматизація та комп'ютерно-інтегровані технології», «Телекомунікація та радіотехніка» усіх форм навчання вищих навчальних закладів

Відповідальний за випуск Зуєв А.О.
Роботу до друку рекомендував Дудник О.В.

План 2022 р., Поз.344
Підписано до друку 05.11.2022. формат 60×84 1/16. Папір друк. № 2.
Друк – різографія. гарнітура Times New Roman. Розум. друк. арк.3,2.
Обл. – вид. арк. 2,7.Наклад 100 прим. Зам. № . Ціна договірна.

Самостійне видання