Київський національний університет імені Тараса Шевченка

Ю.О. Гришко, О.С. Шкільняк

АЛГОРИТМИ ОБЧИСЛЮВАЛЬНОЇ ГЕОМЕТРІЇ

Навчальний посібник

Київ 2020

УДК 004.421

Гришко Ю.О., Шкільняк О.С. Алгоритми обчислювальної геометрії: Навчальний посібник для студентів факультету комп'ютерних наук та кібернетики. – 153 с.

Рецензенти:

Крак Ю.В., доктор фіз.-мат. наук, професор,
завідувач кафедри теоретичної кібернетики
Київського національного університету імені Тараса Шевченка

Дорошенко А.Ю., доктор фіз.-мат. наук, професор,
провідний науковий співробітник Інституту програмних систем НАН України

Рекомендовано до друку вченою радою
факультету комп'ютерних наук та кібернетики
10 лютого 2020 року
протокол № 7

В посібнику наводяться основні поняття обчислювальної геометрії, розглядається постановка основних задач та підходи і методи їх розв'язання. Зокрема, вивчаються алгоритми, присвячені розв'язанню задач геометричного пошуку, побудови опуклої оболонки, близькості та перетину.

Посібник призначений головним чином для студентів спеціальності "Програмна інженерія", а також для широкого кола зацікавлених.

ЗМІСТ

РОЗДІЛ 1
ВСТУП ДО ОБЧИСЛЮВАЛЬНОЇ ГЕОМЕТРІЇ. ІСТОРИЧНЕ ПІДҐРУНТЯ ТА БАЗОВІ ПОНЯТТЯ

1.1 Становлення обчислювальної геометрії

Фундаментом математичної думки можна вважати єгипетську та грецьку геометрії. Одним із основоположників античної геометрії по праву вважається Евклід: саме йому належить найвідоміше з давніх часів застосування аксіоматичного методу, а евклідова побудова наочно демонструє приклад конструктивного доведення. Трактат Евкліда «Начала» (300 рік до н.е.) часто називають найуспішнішим та найвпливовішим підручником всіх часів.

Евклідова побудова включає в себе:

- набір допустимих інструментів (лінійка, циркуль);
- множину допустимих операцій (примітивів), які можна виконати з їх допомогою.

Важливим є питання повноти системи евклідових примітивів за умови скінченності числа операцій (чи можна за її допомогою виконати всі можливі геометричні побудови – зокрема, трисекцію кута).

При цьому античні мислителі розглядали також альтернативні варіанти моделей обчислень, отримані зміною набору примітивів та/або допустимих інструментів. Наприклад, Архімед запропонував (коректну) конструкцію для задачі трисекції кута у 60°, додавши новий примітив. Іноді вивчався обмежений набір інструментів (наприклад, допускався тільки циркуль).

Неповнота системи евклідових примітивів була доведена лише за допомогою алгебри. Нові формулювання геометрії запропонував Рене Декарт, який, ввівши систему координат, зумів виразити геометричні задачі алгебраїчно і перейти до вивчення плоских кривих вищих порядків і ньютонівського аналізу.

Опис геометричних об'єктів за допомогою алгебраїчних рівнянь розширив можливості їх дослідження і поставив нові питання обчислювальності. Ці питання вивчали, зокрема, Н.Г. Абель і К.Ф. Гаусс.

Ще в 1672 році Джордж Мор показав, що будь-яку побудову, яка здійснюється за допомогою циркуля і лінійки, можна виконати лише циркулем у випадку, коли шукані об'єкти визначаються точками (тобто можна промоделювати операції лінійки циркулем).

Оскільки евклідові побудови для нетривіальних задач очевидно трудомісткі, популярною розвагою серед геометрів стало покращення цих побудов в плані зменшення кількості виконаних "операцій".

Однак задача визначення кількісної міри складності була сформульована лише у 1902 році французьким цивільним інженером і математиком Емілем Лемуаном. Систематизація евклідових примітивів Лемуаном:

- поставити одну ніжку циркуля в дану точку,
- поставити одну ніжку циркуля на дану пряму,
- провести коло,
- сумістити ребро лінійки з даною точкою,
- провести лінію.

Загальна кількість таких операцій – *простота побудови*.

Це визначення близьке до поняття часової ефективності алгоритму, але Лемуан нічого не говорить про функціональний зв'язок між розміром вхідних даних (числом заданих точок і ліній) для геометричної побудови та її простоти.

Хоч Лемуану вдалося покращити евклідів розв'язок задачі про кола Аполлонія з 508 кроків до менш ніж 200, ідея про нижні оцінки складності алгоритму до нього не прийшла.

Задача Аполлонія (сформульована Аполлонієм Пергським в 220 р. до н.е., відновлена Вієтом в 1600 р.). Побудувати за допомогою циркуля і лінійки коло, дотичне до трьох заданих кіл (рис. 1).

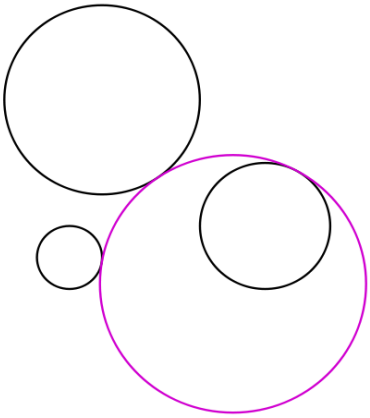


Рисунок 1. Задача Аполлонія

Д. Гільберт на обмеженій моделі вивчав побудови, які можна виконати за допомогою односторонньої лінійки і шкали (зрозуміло, цей набір інструментів можливо використати не для всіх евклідових побудов), і для таких побудов, розглядаючи координати побудованих точок як деяку функцію F від заданих точок, Гільберт сформулював необхідну та достатню умови обчислюваності F з використанням рівно n операцій добування квадратного кореня.

Лемуан й інші математики вивчали не лише часову складність евклідових побудов, але й підіймали питання про об'єм простору, необхідний для цих побудов.

Незважаючи на те, що не всі геометричні побудови можна виконати з допомогою евклідових примітивів, наближені асимптотично збіжні процедури були відомі ще з античності (квадратура круга, подвоєння куба).

Обчислювальна геометрія – це наука, предметом дослідження якої є аналіз та побудова ефективних алгоритмів розв'язання геометричних задач та оцінки їх складності. Вона має багато сфер застосування, серед них:

- комп'ютерна графіка,
- математична візуалізація,
- системи автоматизованого проектування і розрахунку,
- робототехніка (motion planning, видимість),
- геоінформаційні системи (геометричний пошук і локалізація, планування маршруту),
- дизайн мікросхем,
- комп'ютерний зір (3D-реконструкція).

Обчислювальна геометрія включає в себе два окремі напрями.

1. Комбінаторна обчислювальна геометрія (алгоритмічна геометрія).
2. Чисельна обчислювальна геометрія (геометричне моделювання – моделювання і представлення кривих та поверхонь).

Основні класи задач обчислювальної геометрії:

- опуклість;
- перетин;
- геометричний пошук;
- близькість;
- оптимізація.

За способом обробки:

- пошук підмножини;
- обчислення;
- розпізнавання.

1.2 Оцінка ефективності алгоритмів. Звідність задач

Аналіз алгоритмів

Для аналізу алгоритмів використовують наступні показники:

- *часова ефективність* (time efficiency) – індикатор швидкості виконання алгоритму;

- *просторова ефективність* (space efficiency) – кількість додаткової оперативної пам'яті, необхідної для виконання алгоритму.

Граничну поведінку складності при збільшенні розміру задачі називають *асимптотичною ефективністю*.

В якості моделі обчислень будемо використовувати *однопроцесорну машину з довільним доступом до пам'яті* (random-access machine, RAM). Наведемо її основні особливості.

В кожній комірці пам'яті може зберігатися єдине дійсне число.

Вбудовані елементарні команди:

– арифметичні (додавання, віднімання, множення, ділення, остача від ділення, операції округлення);

– операції порівняння чисел;

– операції переміщення даних в пам'яті;

– механізм непрямої адресації пам'яті (цілочисельні адреси);

– управляючі конструкції (розгалуження, цикл, механізм виклику підпрограми).

За необхідності використовуються додаткові логічні, алгебраїчні та тригонометричні операції.

Виконання операцій послідовне. Кожна елементарна операція має фіксовану вартість (іноді для простоти вважають вартість одиничною).

Ефективність алгоритмів

• Ефективність алгоритму в найкращому випадку (best-case efficiency): ефективність для найкращих вхідних даних; найменша можлива кількість операцій.

• Ефективність алгоритму в найгіршому випадку (worst-case efficiency): ефективність для найгірших вхідних даних; максимальна міра ефективності даного алгоритму для всіх задач заданого розміру – верхня оцінка швидкодії.

• Ефективність алгоритму в середньому випадку (average-case efficiency): ефективність для довільних вхідних даних; "середня" складність за всіма даними заданого розміру.

О-позначення

Асимптотична верхня границя: для даної функції $g(n)$ це множина функцій $\mathrm{O}\big(g(n)\big)$, таких що:

$$\mathrm{O}\big(g(n)\big) = \begin{cases} f(n): \text{існують додатні константи } c \text{ та } n_0, \\ \quad \text{що } 0 \le f(n) \le cg(n) \text{ для всіх } n \ge n_0. \end{cases}$$

Це верхня межа функції $f(n)$ з точністю до константи.

В принципі, при $f(n) = O(g(n))$ не важливо наскільки близько $f(n)$ буде знаходитися до своєї верхньої границі (рис. 2).

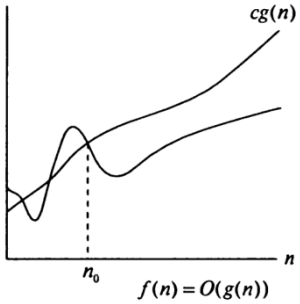


Рисунок 2. O-позначення

Ω-позначення

Асимптотична нижня границя: для даної функції $g(n)$ це множина функцій $\Omega(g(n))$, таких що:

$$\Omega(g(n)) = \begin{cases} f(n): \text{існують додатні константи } c \text{ та } n_0, \\ \text{що } 0 \le cg(n) \le f(n) \text{ для всіх } n \ge n_0. \end{cases}$$

Це нижня межа функції $f(n)$ з точністю до константи (рис. 3).

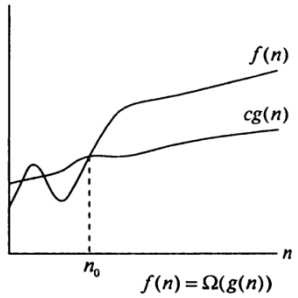


Рисунок 3. Ω-позначення

Θ-позначення

Асимптотична точна оцінка: для даної функції $g(n)$ це множина функцій $\Theta(g(n))$, таких що:

$$\Theta(g(n)) = \begin{cases} f(n): \text{існують додатні константи } c_1, c_2 \text{ та } n_0, \\ \text{що } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ для всіх } n \ge n_0. \end{cases}$$

Тобто, якщо $f(n) = \Theta(g(n))$, то $f(n)$ можна «вставити» між функцій $c_1 g(n)$ та $c_2 g(n)$ при досить великих n (рис. 4).

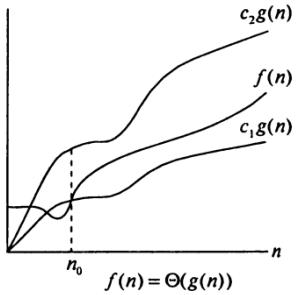


Рисунок 4. Θ-позначення

Теорема 1. Для будь-яких функцій $f(n)$ та $g(n)$ співвідношення $f(n) = \Theta\big(g(n)\big)$ виконується тоді і тільки тоді, коли $f(n) = \mathrm{O}\big(g(n)\big)$та $f(n) = \Omega\big(g(n)\big)$.

Теорему головним чином використовують для отримання асимптотично точної оцінки за допомогою асимптотичних верхньої та нижньої границь.

Теорема 2. Нехай $t_1(n) = \mathrm{O}\big(g_1(n)\big)$, $t_2(n) = \mathrm{O}\big(g_2(n)\big)$.

Тоді $t_1(n) + t_2(n) = \mathrm{O}(max\{g_1(n), g_2(n)\})$.

Зауваження. Аналогічні твердження будуть справедливі і для множин Ω та Θ.

Отже, загальна ефективність алгоритму залежить від тієї його частини, яка має найбільший порядок зростання, тобто найменш ефективної його частини.

Оцінки складності деяких задач

Якщо деяка задача в d-вимірному просторі вимагає $f(n)$ часу, то для випадку k-вимірного простору ($k < d$) буде необхідно як мінімум також $f(n)$ часу.

Перевірка належності елемента *n*-елементній множині: час $\Omega(\log_2 \mathrm{n})$.

Складність порядку $\Omega(n \log_2 \mathrm{n})$:

- сортування множини з *n* елементів,
- перевірка унікальності елементів множини,
- перевірка на значимість: задана множина n точок на площині, чи всі точки належать опуклій оболонці множини точок,
- перевірка ε-близькості: задана множина з n точок на площині, чи є серед елементів цієї множини два елементи, які знаходяться на відстані ε,
- об'єднання інтервалів.

Перетворення (звідність) задач

Кажуть, що задача A може бути *перетворена* в задачу B (або: задача A *зводиться* до задачі B), якщо:

1. вхідні дані задачі A перетворюються у вхідні дані задачі B,

2. розв'язується задача B,

3. результат розв'язання задачі B перетворюється у правильний розв'язок задачі A.

Якщо кроки 1 та 3 можна виконати за час $O(\tau(n))$, де N – розмір задачі A, то кажуть, що задача A є $\tau(n)$-звідною до B: $A \propto_{\tau(n)} B$.

Звідність не є симетричним відношенням. Якщо задачі A та B взаємоперетворювані, то вони називаються *еквівалентними*.

Твердження 1 (нижні оцінки методом перетворення). Якщо відомо, що задача A вимагає $T(N)$ часу і $A \propto_{\tau(n)} B$, то B можна розв'язати за час, не менший за $T(N) - O(\tau(n))$.

Твердження 2 (верхні оцінки методом перетворення). Якщо задачу B можна розв'язати за час $T(N)$ часу і $A \propto_{\tau(n)} B$, то A можна розв'язати за час, який не перевищує $T(N) + O(\tau(n))$ (рис. 5).

Рисунок 5. Верхня та нижні оцінки перетворення

1.3 Особливості представлення даних

1.3.1 Представлення даних та робота з динамічними множинами

Рисунок 6. Базові структури даних

Абстрактні типи даних: список, дерево, граф, стек, черга, черга з пріоритетами, дек, множина, асоціативний масив (словник), мультимножина.

Операції над множинами

Нехай ми маємо динамічну скінченну множину S та елемент універсальної множини u. Тоді при роботі з множинами необхідні наступні операції:

- ПРИНАЛЕЖНІСТЬ (u, S). Чи вірно, що $u \epsilon S$?
- ВСТАВИТИ (u, S). Включити u в S.
- ВИЛУЧИТИ (u, S). Вилучити u із S.

Для розбиття, що складається з множин $\{S_1, \dots, S_n\}$, корисні операції:

- ЗНАЙТИ (u). Знайти таке j, що $u \epsilon S_j$.
- ОБ'ЄДНАТИ $(S_i, S_j; S_k)$. Сформувати об'єднання S_i та S_j і назвати його S_k.

Якщо універсальна множина цілком упорядкована, то важливими будуть операції:

- $MIN(S)$. Знайти найменший елемент S.
- РОЗЧЕПИТИ (u, S). Розділити S на $\{S_1, S_2\}$, такі, що $S_1 = \{v: v \epsilon S \text{ і } v \leq u\}$ та $S_2 = S - S_1$.
- ЗЧЕПИТИ (S_1, S_2). Нехай для будь-яких $u_1 \epsilon S_1$ і $u_1 \epsilon S_2$ маємо $u_1 \leq u_2$; необхідно сформувати множину $S = S_1 \cup S_2$.

Абстрактні типи даних можна класифікувати відповідно до допустимих на них операцій. Приклади абстрактних типів даних:

Для впорядкованих множин:

- Словник: НАЛЕЖНІСТЬ, ВСТАВИТИ, ВИЛУЧИТИ
- Пріоритетна черга: MIN, ВСТАВИТИ, ВИЛУЧИТИ
- Зчеплена черга: ВСТАВИТИ, ВИЛУЧИТИ, РОЗЧЕПИТИ, ЗЧЕПИТИ

Для невпорядкованих множин:

- Піраміда злиття: ВСТАВИТИ, ВИЛУЧИТИ, ЗЛИТИ, MIN

В геометричних задачах часто використовуються специфічні модифікації структур даних, основними з яких є *дерево відрізків* та *реберний список з подвійними зв'язками*.

1.3.2 Дерево відрізків

Структура даних, створена для роботи з інтервалами на числовій осі, кінці яких належать *фіксованій* множині із N абсцис (ці абсциси можна нормалізувати, замінюючи кожну із них її порядковим номером при обході їх зліва направо). Можна вважати абсциси цілими числами на інтервалі $[1, N]$.

Дерево відрізків – це двійкове дерево з коренем $T(l, r)$, що будується рекурсивно для заданих цілих чисел l і r таких, що $l < r$ наступним чином (див. рис. 7).

Дерево складається із кореня v з параметрами $B[v] = l$ та $E[v] = r$ (смисл: $B - Begin$, $E - End$). Якщо $r - l >$, то воно складається із лівого піддерева $T(l, [(B[v] + E[v])/2])$ та правого піддерева $T([(B[v] + E[v])/2], r)$. Корені піддерев природно позначити через ЛСИН$[v]$ та ПСИН$[v]$ відповідно.

Параметри $B[v]$ та $E[v]$ позначають інтервал $\left[B[v], E[v]\right] \subseteq [l, r]$, зв'язаний з вузлом v. Інтервали, що належать множині $\{B[v], E[v]: v - \text{вузол } T(l, r)\}$ – *стандартні інтервали* дерева $T(l, r)$.

Стандартні інтервали, що належать листам $T(l, r)$ – *елементарні інтервали*. Дерево $T(l, r)$ збалансоване і має глибину $[\log_2(r - l)]$. Строго кажучи, інтервал, пов'язаний з вузлом v, є напіввідкритим: $\left[B[v], E[v]\right]$, крім вузлів на найправішому шляху в дереві, інтервали яких будуть замкнутими.

Дерево відрізків $T(l, r)$ призначене для *динамічного* зберігання відрізків з кінцями, що належать до множини $(l, l_1, \dots, r)$. Тобто допустимими операціями є вставка та видалення.

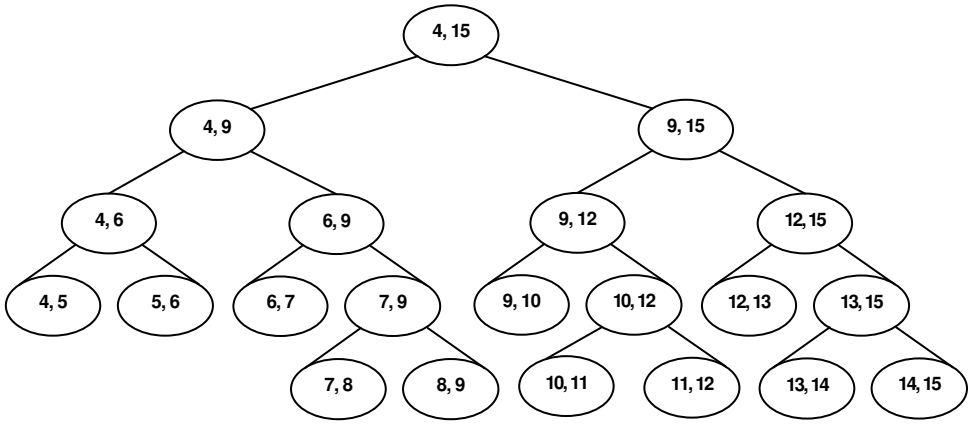


Рисунок 7. Приклад дерева відрізків $T(4{,}15)$

Вставка в дерево відрізків

Фрагментація інтервалу $[b, e]$ цілком визначається операцією, яка вставляє $[b, e]$ в дерево відрізків T (рис. 8).

procedure ВСТАВИТИ$\big(b, e;$ корінь$(T)\big)$
begin **if** $b \le B(v)$ **and** $E(v) \le e$ **then** покласти $[b, e]$ вузлу значення v
else **begin** **if** $\big(b < \left[B[v] + E[v]\right]/2\big)$ **then**
ВСТАВИТИ$(b, e;$ ЛСИН$[v])$;
if $\left(\left[B[v] + E[v]\right]/2 < e\right)$ **then**
ВСТАВИТИ$(b, e;$ ПСИН$[v])$
end
end.

Дія ВСТАВИТИ$\big(b, e;$ корінь$(T)\big)$ відповідає «маршруту» в T, який має загальну структуру:

– початковий шлях $P_{\text{поч}}$ від кореня до вузла v^* (*розгалуження*), із якого виходять два шляхи – $P_{\text{Л}}$ та $P_{\text{П}}$;
– інтервал, що вставляється, відноситься або повністю до розгалуження, або до усіх правих синів шляху $P_{\text{Л}}$ та всіх лівих синів шляху $P_{\text{П}}$; при цьому визначається фрагментація $[b, e]$ (*вузли віднесення*).

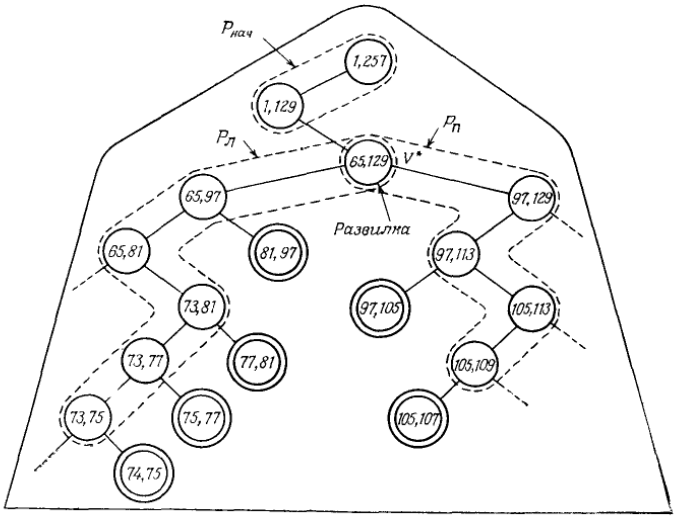


Рисунок 8. Вставка інтервалу $[74, 107]$ в $T(1{,}257)$

Існує декілька способів організації віднесення інтервалу до вузла v дерева T.

Здебільшого достатньо знати лише кількість інтервалів, що відноситься до даного вузла v. Її можна заносити в параметр $\mathrm{C}[v]$. Тоді віднесення відрізка $[b, e]$ до вузла v виглядатиме як $\mathrm{C}[v] := \mathrm{C}[v] + 1$.

Щоб зберігати інформацію про інтервали, віднесені до даного вузла, до кожного вузла додається список ідентифікаторів таких інтервалів.

Видалення з дерева відрізків

Алгоритм видалення симетричний вставці:

procedure ВИДАЛИТИ$(b, e; v)$
begin **if** $b \le B(v)$ **and** $E(v) \le e$ **then** $C[v] := C[v] - 1$
else **begin** **if** $\left(b < \left[B[v] + E[v]\right]/2\right)$ **then**
ВИДАЛИТИ$(b, e;$ ЛСИН$[v])$;
if $\left(\left[B[v] + E[v]\right]/2 < e\right)$ **then**
ВИДАЛИТИ$(b, e;$ ПСИН$[v])$
end
end.

Вважаємо, що нас цікавить лише параметр $C[v]$.

Зауваження. Процедура коректно працюватиме за умови видалення раніше вставлених відрізків.

Інший спосіб побудови дерева відрізків

Дерево відрізків є червоно-чорним деревом, кожен елемент якого містить відрізок $int[x]$, що визначається своїми кінцями. Ключем вузла є лівий кінець відрізка $low[int[x]]$. Таким чином, симетричний обхід дає перелік відрізків, відсортованих по лівих кінцях.

Додатково вузли містять значення $max[x]$ – максимальне значення з правих кінців всіх відрізків, що зберігаються в піддереві з коренем x. Це значення може визначатися через дочірні вузли так:

$$max[x] = max(high[int[x]], max[left[x]], max[right[x]]).$$

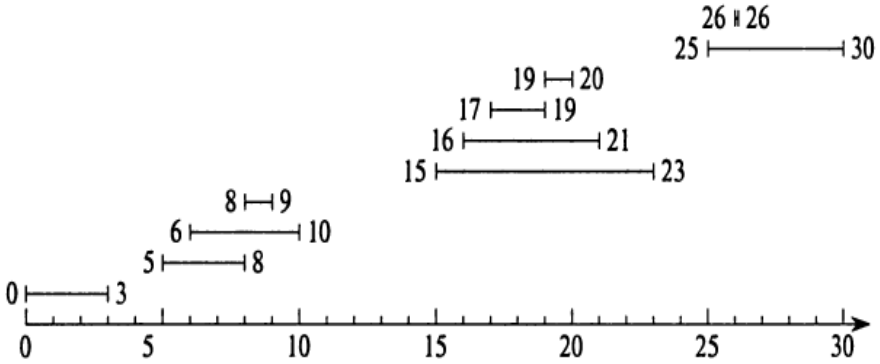


Рисунок 9.

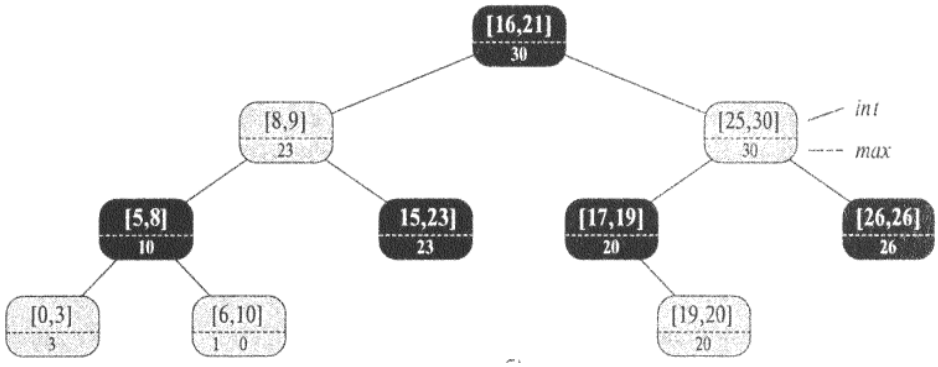


Рисунок 10.

Операції над деревом:

- вставка елемента (стандартна);
- видалення елемента (стандартне);

- пошук відрізка в дереві T, що перекривається з заданим відрізком i (нова операція):

$Interval\_Search(T, i)$
1 $x \leftarrow root[T]$
2 **while** $x \neq nil[T]$ та i не перекривається з $int[x]$
3 **do if** $left[x] \neq nil[T]$ та $max\big[left[x]\big] \geq low[i]$
4 **then** $x \leftarrow left[x]$
5 **else** $x \leftarrow right[x]$
6 **return** x

Всі операції виконуються за час $O(\log_2 n)$.

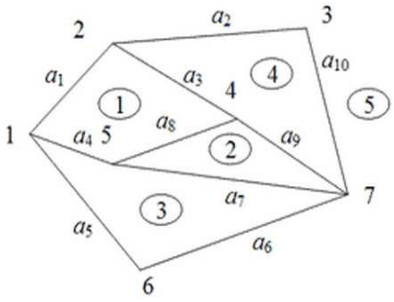
1.3.3 Реберний список з подвійними зв'язками (РСПЗ)

Призначений для представлення планарних графів, укладених на площині (тобто плоских).

Нехай в планарному графі (V, E) маємо $V = \{v_1, \dots, v_n\}$ та $E = \{e_1, \dots, e_m\}$. Головна компонента РСПЗ – *реберний вузол*, який містить чотири інформаційних поля (V_1, V_2, F_1, F_2) і два поля вказівників (P_1, P_2):

- V_1 містить початок ребра, V_2 – кінець ребра (так ребро отримує умовну орієнтацію),
- F_1 та F_2 містять імена граней, які лежать відповідно ліворуч і праворуч від ребра, орієнтованого від V_1 до V_2,
- Вказівник P_1 (відповідно P_2) задає реберний вузол, який містить перше ребро, що зустрічається слідом за ребром (V_1, V_2) при повороті від нього проти годинникової стрілки навколо V_1 (відповідно V_2).

Імена граней і вершин можуть бути задані цілими числами (рис. 11).



	V_1	V_2	F_1	F_2	P_1	P_2
a_1	1	2	5	1	5	3
a_2	2	3	5	4	1	10
a_3	2	4	4	1	2	8
a_4	1	5	1	3	1	7
a_5	1	6	3	5	4	6
a_6	6	7	3	5	5	10
a_7	5	7	2	3	8	6
a_8	5	4	1	2	4	9
a_9	4	7	4	2	3	7
a_{10}	3	7	5	4	2	9

Рисунок 11. Приклад графа, представленого РСПЗ

Якщо граф має N вершин та F граней, можна ввести масиви $HV[1:N]$ та $HF[1:F]$, що містять номери ребер, пов'язаних з відповідною вершиною чи гранню. Процедура їх заповнення виглядатиме так:

```
for i ← 1 to N do
begin
        if (HV[V1[i]] = 0) then HV[V1[i]] ← i;
           if (HV[V2[i]] = 0) then HV[V2[i]] ← i;
                if (HF[F1[i]] = 0) then HF[F1[i]] ← i;
                   if (HF[F2[i]] = 0) then HF[F2[i]] ← i;
end;
```

Для зображеного графа масиви HV та HF набудуть значень: $HV = [1,1,2,3,4,5,6]$, $HF = [1,7,4,2,1]$.

За допомогою масива HV та процедури ВЕРШИНА(j) ми можемо перелічити проти годинникової стрілки всі ребра, що виходять з вершини v_j. Процедура будує послідовність ребер, інцидентних v_j, як послідовність адрес, занесених в масив A.

```
        procedure ВЕРШИНА(j)
begin   a:= HV[j];
        a0:= a;   A[1]:= a;
        i:= 2;
        if (V1[a] = j) then a:= P1[a] else a:= P2[a];
           while (a ≠ a0) do
              begin  A[i]:= a;
                   if (V1[a] = j) then a:= P1[a] else a:= P2[a];
                   i:= i + 1
              end
end.
```

Час роботи пропорційний кількості ребер інцидентних v_j.

Аналогічно можна описати процедуру побудови послідовності ребер, які обмежують грань f_j. Для цього в попередній процедурі досить замінити $(HV, V1)$ на $(HF, F1)$. При цьому ребра тепер будуть перелічуватися за годинниковою стрілкою навколо грані.

Якщо граф спочатку заданий списком інцидентності, де з кожною вершиною співвіднесено список інцидентних їй ребер, перелічених в порядку обходу проти стрілки годинника навколо цієї вершини, то можна перейти до РСПЗ за час $O(|V|)$.

РОЗДІЛ 2
ГЕОМЕТРИЧНИЙ ПОШУК

2.1 Вступ до геометричного пошуку

d-вимірний евклідів простір E^d – простір d-плексів $(x_1, x_2, \ldots, x_d)$, що складаються з дійсних чисел x_i $(i = 1, \ldots, d)$ з відстанню $\sqrt{\sum_{i=1}^{d} x_i^2}$.

Точка p в просторі E^d – d-плекс $(x_1, x_2, \ldots, x_d)$.

Пряма в просторі E^d: лінійна комбінація $aq_1 + (1 - a)q_2$ $(a \in \boldsymbol{R})$, де q_1, q_2 – різні точки.

Прямолінійний відрізок $\overline{q_1 q_2}$: опукла комбінація $aq_1 + (1 - a)q_2$ $(a \in [0; 1])$, де q_1, q_2 – різні точки.

Опукла оболонка множини точок S простору E^d – границя найменшої опуклої області в E^d, що охоплює S.

Многокутник в просторі E^2 – скінченна множина відрізків, в якому кожний кінець відрізка належить рівно двом відрізкам і жодна підмножина відрізків не має вказаної властивості. Ці відрізки називаються сторонами (ребрами), а їх кінці – вершинами многокутника. Часом під терміном «многокутник» розуміють об'єднання границі та внутрішньої області.

Многокутник P *простий*, якщо жодна пара непослідовних його ребер не має спільних точок. Простий многокутник розбиває площину на дві області – внутрішню (скінченну) та зовнішню (нескінченну).

Простий многокутник P *опуклий*, якщо його внутрішня область є опуклою множиною.

Простий многокутник P *зірковий*, якщо існує точка z, не зовнішня для P, така, що для всіх точок p, що належать P, відрізок zp повністю лежить всередині P. Множина точок z, яка має вказану властивість – ядро P. Опуклий многокутник співпадає зі своїм ядром (рис. 12).

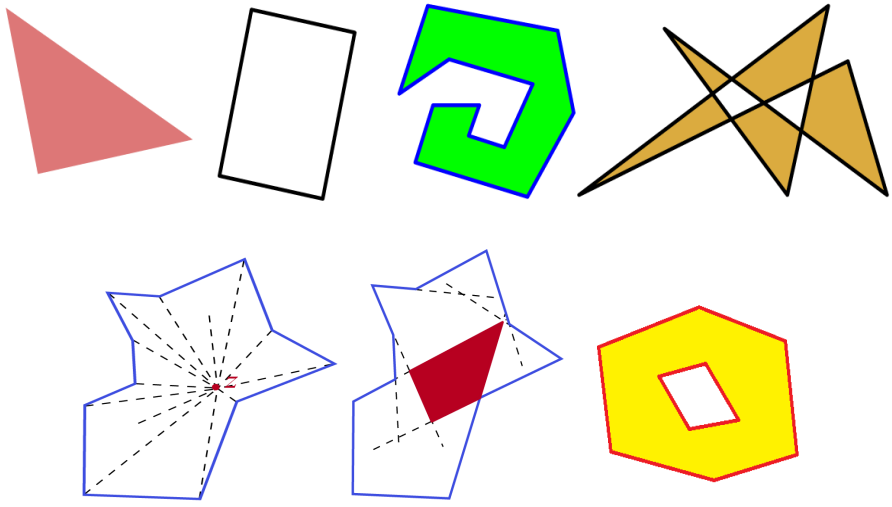


Рисунок 12. Приклади многокутників

Вступ до геометричного пошуку

Загальне формулювання задачі пошуку.

Дано деякий набір даних («файл») і деякий новий елемент даних («зразок»). Пошук – встановлення зв'язку між зразком і файлом. Він зводиться до визначення позиції відповідного елемента в заданому наборі даних (Д. Кнут).

Особливості *геометричного пошуку*:

- в геометричних застосуваннях «файли» відображають складніші структури (многокутники, поліедри і т.д.);
- результатом пошуку може бути не елемент файлу, який відповідає зразку, а скоріше положення останнього відносно файлу.

Пошуковий запит: пошукове повідомлення, відповідно до якого ведеться перегляд файлу. Його організація сильно залежить від типу файлу. Від набору допустимих запитів залежать і алгоритми їх обробки.

Нехай є набір геометричних даних і необхідно дізнатись, чи мають вони певну властивість (скажімо, опуклість). У найпростішому випадку таке питання виникає один раз. Запит такого типу називається *унікальним*. Запити, обробка яких повторюється багатократно на тому ж самому файлі, називаються *масовими*.

Передобробка: процес розташування даних у зручній для подальшої обробки структурі даних.

Для ефективності пошуку інформацію зручно розташовувати відповідно до деякої структури. При цьому витрачається певний ресурс. Аналіз ефективності алгоритму зосереджується на чотирьох основних мірах його оцінки:

1. Час пошуку. (Скільки часу треба як в середньому, так і в гіршому випадку для відповіді на один запит?)
2. Пам'ять. (Скільки необхідно пам'яті для структури даних?)
3. Час попередньої обробки. (Скільки часу необхідно для організації даних перед пошуком?)
4. Час коригування. (Вказано елемент даних. Скільки треба часу на його включення до структури даних або вилучення з неї?)

Різниці в затратах по всім мірам ефективності при різних підходах до вирішення задачі розглянемо на прикладі розв’язання наступної задачі регіонального пошуку:

П1. РЕГІОНАЛЬНИЙ ПОШУК-ПІДРАХУНОК

Дано N точок на площині. Скільки із них лежить усередині заданого прямокутника, сторони якого паралельні координатним осям? Тобто, скільки точок (x, y) задовольняють нерівностям $a \le x \le b$, $c \le y \le d$ для заданих a, b, c і d (рис. 13).

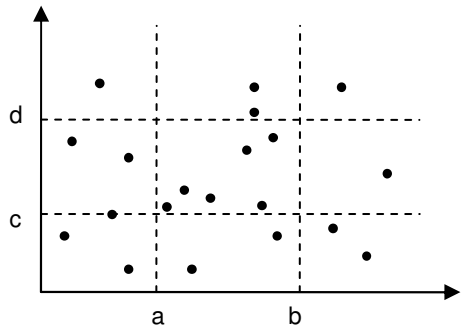


Рисунок 13. Регіональний пошук-підрахунок

Очевидно, унікальний запит виконується за лінійний час: кожна з N точок перевіряється на задовольняння нерівностей, що задають прямокутник.

Витрати за пам'яттю теж будуть лінійними, оскільки треба запам'ятати лише $2N$ координат. Відсутні витрати на попередню обробку. Час коригування нової точки константний.

У випадку масових запитів для розв’язання задачі скористаємося комбінацією методів локусів та векторного домінування.

Метод локусів

У геометричних задачах запиту ставиться у відповідність точка в зручному для пошуку просторі, а цей простір розбивається на області (локуси), у межах яких відповідь не змінюється. Тобто, якщо вважати еквівалентними два запити, на які отримують однакові відповіді, то кожна область розбиття простору пошуку відповідає одному класу еквівалентності запитів.

Для розглянутої задачі можна замінити запит з прямокутником чотирма підзадачами, по одній на кожну із його вершин, і сумістити їх розв'язки для одержання остаточної відповіді.

Підзадача, пов'язана з вершиною p, полягає у визначенні кількості точок $Q(p)$ заданої множини, які задовольняють нерівностям $x \leq x(p)$ і $y \leq y(p)$, тобто кількості точок у лівому нижньому квадранті, визначеному p (рис. 14):

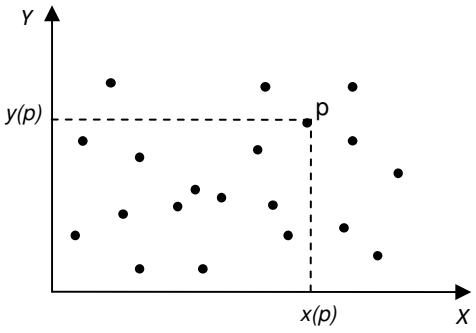


Рисунок 14. Скільки точок "південно-західніше" p?

Векторне домінування

Точка (вектор) v домінує над w, якщо для всіх i виконується $v_i \geq w_i$.

На площині точка v домінує над $w \Leftrightarrow w$ лежить в лівому нижньому квадранті, який визначається v.

Розглянемо зв'язок між домінуванням та регіональним пошуком.

Нехай $Q(p)$ – число точок, над якими домінує p.

Тоді число точок $N(p_1p_2p_3p_4)$, які належать прямокутнику $p_1p_2p_3p_4$, визначається як $N(p_1p_2p_3p_4) = Q(p_1) - Q(p_2) - Q(p_4) + Q(p_3)$ (рис. 15).

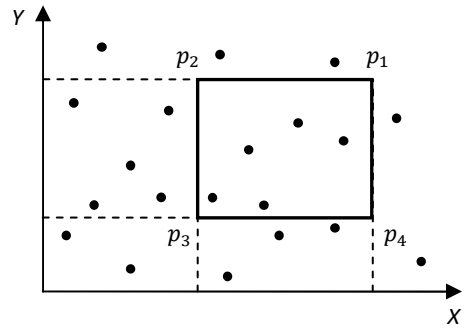


Рисунок 15. $N(p_1p_2p_3p_4) = Q(p_1) - Q(p_2) - Q(p_4) + Q(p_3)$

Таким чином, задача регіонального пошуку зводиться до задачі обробки запитів про домінування для чотирьох точок.

Властивість, яка спрощує ці запити: на площині існують області зручної форми, всередині яких число домінування Q є константою.

Припустимо, що із наших точок на осі ***X*** та ***Y*** опущено перпендикуляри, а отримані лінії продовжені у нескінченність. Вони утворюють решітку із $(N+1)^2$ прямокутників (рис. 16):

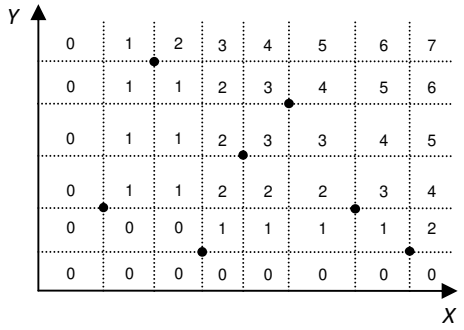


Рисунок 16. Решітка із $(N+1)^2$ прямокутників

Для усіх точок p у кожній такій комірці $Q(p)$ є константою.

Це означає, що домінантний пошук є відповіддю на питання: у якій комірці прямокутної решітки лежить задана точка?

Після впорядкування початкових точок за обома координатами залишається виконати два двійкових пошуки (по одному для кожної осі), щоб знайти комірку, яка містить шукану точку. Час запиту буде рівним $O(\log N)$. Маємо $O(N^2)$ комірок, тому необхідна квадратична пам'ять.

При нашому способі, визначення числа домінування для будь-якої комірки можна зробити за час $O(N)$, що призводить до загальної витрати часу $O(N^3)$ на попередню обробку (втім, його можна зменшити до $O(N^2)$) (табл.1).

Метод	**Запит**	**Пам'ять**	**Передобробка**
Локусів і векторного домінування	$O(\log N)$	$O(N^2)$	$O(N^2)$
Дерева регіонів	$O(\log^2 N)$	$O(N \log N)$	$O(N \log N)$
Без попередньої обробки	$O(N)$	$O(N)$	$O(N)$

Таблиця 1. Порівняння ефективності різних методів регіонального пошуку

Основні моделі задач геометричного пошуку:

1. *Задачі локалізації.* Файл є розбиттям геометричного простору на області, а запитом є точка. Локалізація полягає у визначені області, яка містить шукану точку.

2. *Задачі регіонального пошуку*. Файл містить набір точок простору, а запитом є деяка стандартна геометрична фігура, яка довільно переміщується у цьому просторі (типовий запит для тривимірного простору – це куля або брус). Регіональний пошук полягає у визначенні (задача звіту) або у підрахунку кількості (задача підрахунку) всіх точок всередині заданого регіону (області).

2.2 Задачі локалізації точки

2.2.1 Приналежність точки многокутнику

Ці задачі можна також назвати задачами про приналежність точки. Трудомісткість розв'язання цієї задачі суттєво залежить від природи простору і способу його розбиття.

Наразі плоска задача добре вивчена, про випадок E^3 відомо менше, і ще менше – про випадки більшої розмірності.

Розглянемо розбиття площини, або планарні підрозбиття, утворені прямолінійними відрізками. Можна сформулювати такі найпростіші задачі:

П2. ***ПРИНАЛЕЖНІСТЬ ПРОСТОМУ МНОГОКУТНИКУ.*** Дано простий многокутник P і точка z; визначити, чи знаходиться точка z всередині P.

П3. ***ПРИНАЛЕЖНІСТЬ ОПУКЛОМУ МНОГОКУТНИКУ.*** Дано опуклий многокутник P і точка z; визначити, чи знаходиться точка z всередині P.

Наведемо алгоритм розв'язку для випадку *унікального* запиту. Цей результат поширюється також на неопуклі многокутники.

Теорема 1. Приналежність точки z внутрішній області простого N-кутника P можна встановити за час $O(n)$ без передобробки.

Проведемо через пробну точку z горизонталь l (рис. 17).

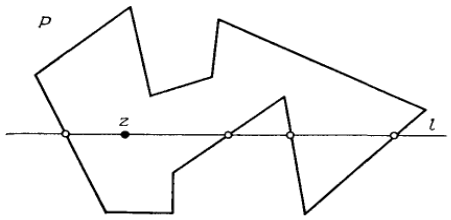


Рисунок 17. Приналежність точки многокутнику

За теоремою Жордана для многокутників, зовнішня і внутрішня області P добре визначені. Варіанти розташування прямої відносно P:

1. Якщо l не перетинає P, то z зовнішня точка.
2. Нехай l перетинає P:
   - при цьому l не проходить через жодну з вершин P. Нехай L – число точок перетину l з границею P ліворуч від z. Точка z лежить всередині $P \Leftrightarrow L$ непарне;
   - вироджений випадок: l проходить через вершини P.

Існує підхід, хоч і не досить чіткий, який дозволяє розглядати вироджений випадок. Від виродженості позбавляємося, здійснивши нескінченно малий поворот l навколо z проти годинникової стрілки, при цьому:

1) якщо обидві вершини ребра належать l, то це ребро відкидається;
2) якщо рівно одна вершина ребра лежить на l, то перетин враховується, коли ця вершина з великою ординатою, і ігнорується в іншому випадку.

Псевдокод алгоритму (час виконання $O(N)$):

begin $L := 0$;
  for $i := 1$ **to** N **do if** ребро (i) не горизонтально
    then
      if (ребро (i) перетинає l нижнім кінцем ліворуч від z)
      then $L := L + 1$;
  if (L непарне) **then** z всередині **else** z зовні
end

Для *масових* запитів розглянемо випадок, коли P – *опуклий* многокутник. Вершини опуклого многокутника впорядковані за полярними кутами відносно довільної внутрішньої точки.

Нехай q – довільна внутрішня точка, z – шукана точка. За точку q можна, наприклад, взяти центр мас (центроїд) трикутника, утвореного будь-якою трійкою вершин P:

$$x_q = (x_{p1} + x_{p2} + x_{p3})/3,\ y_q = (y_{p1} + y_{p2} + y_{p3})/3.$$

Проведемо з точки q промені, що проходять через вершини многокутника. Ці промені розбивають площину на N клинів. Кожен з клинів розбитий на дві частини одним з ребер P, одна частина цілком всередині фігури, інша – цілком ззовні.

Вважаючи q за початок полярних координат, відшукуємо клин, якому належить точка z, а потім перевіряємо, де z знаходиться відносно ребра, що розбиває цей клин (рис. 18).

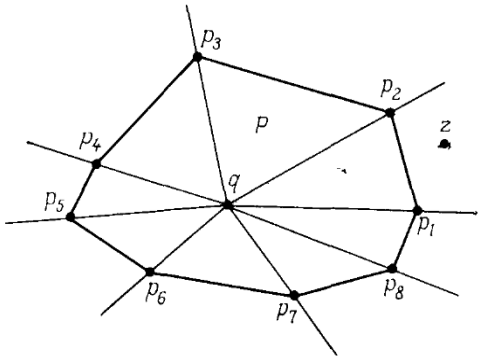


Рисунок 18. Процедура приналежності опуклому многокутнику

Дана пробна точка z. Визначаємо методом бінарного пошуку клин, в якому вона лежить. Точка z лежить між променями, які визначаються p_i і p_{i+1}, тоді і тільки тоді, коли $\angle zqp_{i+1} > 0$ та $\angle zqp_i < 0$ (загалом визначається за час $O(\log N)$).

Якщо p_i і p_{i+1} знайдені, то z – внутрішня точка $\Leftrightarrow \angle p_i p_{i+1} z < 0$ (час $O(1)$). Тобто q та z лежать по один бік від $p_i p_{i+1}$.

Передобробка полягатиме у визначенні внутрішньої точки q та розміщенні послідовності вершин $p_1, \dots, p_N$ у структурі даних, що дозволяє бінарний пошук (час $O(N)$).

Зауваження. Якщо покласти $p_i = (x_i, y_i)$, то визначник

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_2 & 1 \end{vmatrix}$$

дає подвоєну орієнтовану площу трикутника $(p_1p_2p_3)$, де знак "+", коли обхід $(p_1p_2p_3)$ орієнтований проти годинникової стрілки (лівий поворот) і "–", коли обхід $(p_1p_2p_3)$ орієнтований за годинниковою стрілкою (правий поворот).

Теорема 2. Час відповіді на запит про приналежність точки *опуклому* N-кутнику дорівнює $O(logN)$ при витраті $O(N)$ пам'яті і $O(N)$ часу на попередню обробку.

Зірковий многокутник P (рис. 19) містить принаймні одну точку q таку, що $[q, p_i]$ лежить повністю всередині многокутника P для будь-якої вершини p_i із P, $i = 1, \dots, N$. Множина шуканих центрів всередині P є ядром зіркового многокутника (його можна знайти за час $O(N)$).

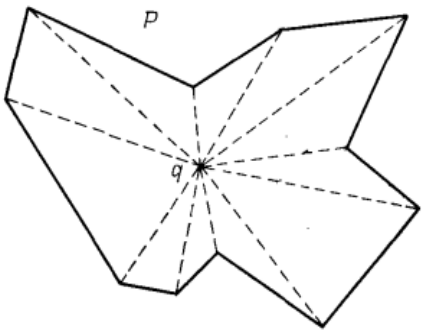


Рисунок 19. Зірковий многокутник P

Теорема 3. Час відповіді на запит про приналежність точки *зірковому* N-кутнику складає $O(logN)$ при витраті $O(N)$ пам'яті і $O(N)$ часу на попередню обробку.

Задача про приналежність точки зірковому многокутнику асимптотично не складніша за її приналежність опуклому многокутнику.

Ієрархія властивостей многокутників:

ОПУКЛІСТЬ ⊂ ЗІРКОВІСТЬ ⊂ ПРОСТОТА.

Однак для випадку приналежності точки простому многокутнику задача вже не така проста. Один з підходів до її вирішення – через розбиття фігури на многокутники спеціального вигляду (опуклі, зіркові чи просто трикутники)

Таким чином, цей підхід означає перетворення N-вершинного простого многокутника в N-вершинний плоский граф. Тому далі будемо розглядати задачу локалізації точки на планарному розбитті.

2.2.2 Локалізація точки на планарному розбитті. Метод смуг

Планарний граф завжди може бути розміщений на площині таким чином, що його ребра стануть прямолінійними відрізками. Графи такого вигляду будуть називатись *плоскими прямолінійними графами* (ППЛГ).

Якщо в такому графі всі степені вершин > 1, то усі області його підрозбиття є простими многокутниками. Без втрати загальності вважатимемо, що розглядаються зв'язні графи.

Головна ідея полягає в тому, щоб створити нові геометричні об'єкти, які дозволяють за допомогою двійкового пошуку локалізувати точку на планарному розбитті.

Розглянемо різні підходи до розв'язання цієї задачі: метод смуг, метод ланцюгів, метод деталізації триангуляції та метод трапецій.

Метод смуг

Нехай задано плоский прямолінійний граф G (рис. 20). Проведемо горизонтальні прямі через кожну його вершину. Вони розділятимуть площину на $(N + 1)$ горизонтальну смугу. Якщо в процесі передобробки провести сортування цих смуг за координатою y, то бінарним пошуком можна знайти ту смугу, в якій лежить точка z, за час $O(\log N)$.

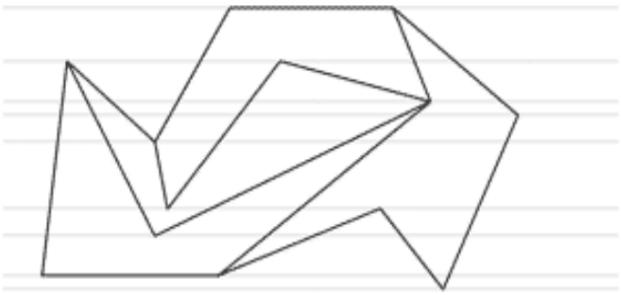


Рисунок 20. Плоский прямолінійний граф

Розглянемо перетин однієї зі смуг із графом G. Перетин складається з відрізків ребер графа G. Ці відрізки визначають трапеції (можуть вироджуватися в трикутники).

Оскільки G планарний граф, то його ребра перетинаються між собою тільки у вершинах, а оскільки кожна вершина лежить на межі смуги, то відрізки ребер всередині смуги не перетинаються. Дані відрізки можна упорядкувати зліва направо й бінарним пошуком знайти ту трапецію, що містить z (рис. 21).

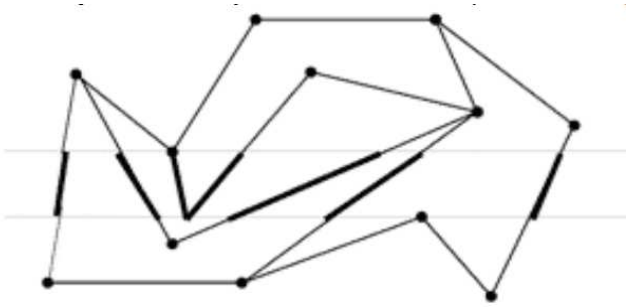


Рисунок 21. Пошук відповідної трапеції

Попередня обробка. Сортуються всі відрізки всередині кожної смуги: $O(N^2 \log N)$ – для часу і $O(N^2)$ – для пам'яті. Час попередньої обробки можна скоротити до $O(N^2)$. Однак існують такі плоскі прямолінійні графи, які потребують квадратичної пам'яті (рис. 22):

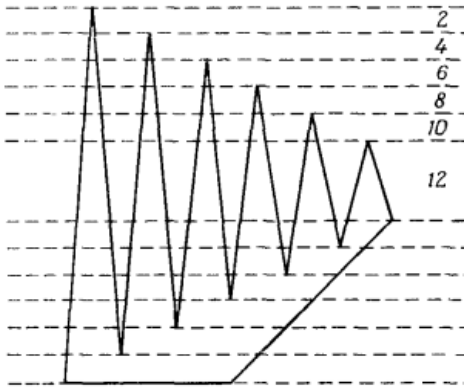


Рисунок 22. Приклад прямолінійного графа, що потребує квадратичної пам’яті при попередній обробці (цифрами позначене число відрізків в смузі).

Розглянемо один з ключових підходів, що має широке застосування в обчислювальній геометрії: *метод плоского замітання (МПЗ).* Уявна (частіше вертикальна) пряма рухається по площині, зупиняючись в деяких точках. Геометричні операції обмежуються геометричними об'єктами, які або перетинаються із замітаючою прямою, або знаходяться в безпосередній близькості від неї в точках її зупинки. Повний розв’язок задачі стає доступним після того, як ця пряма пройде над усіма об'єктами.

Основні структури даних методу плоского замітання:

- список точок подій – послідовність позицій, які призначені для замітаючої прямої;
- статус замітаючої прямої – опис перетину замітаючої прямої з геометричним об'єктом, який замітають.

Скористаємося цим методом для скорочення часу передобробки.

В методі смуг список точок подій – це просто перелік знизу вгору смуг, які відповідають вершинам плоского прямолінійного графу.

Статус – це впорядкована зліва направо послідовність відрізків (ребер графа) всередині розташованої вище смуги. Ця послідовність зберігається в межах однієї смуги.

При переході на нову смугу досягається нова вершина графа v. При цьому зі статусу видаляються ребра, що закінчуються в v, та додаються нові, що виходять з v. Статус коригується в кожній досягнутій точці подій. Статус зручно зберігати в структурі, що дозволяє видалення і вставку елемента за логарифмічний час (збалансовані дерева пошуку).

Витрати ресурсів:

- вставка та вилучення кожного із ребер графа – складність $O(\log N)$ на одну операцію; загальний час $O(N \log N)$, бо в графі $O(N)$ ребер (теорема Ейлера);
- запам'ятовування статусу (час $O(N^2)$ – залежить від кількості відрізків).

Теорема 4. Локалізацію точки в N-вершинному планарному розбиті методом плоского замітання можна реалізувати за час $O(\log N)$ з використанням $O(N^2)$ пам'яті, якщо $O(N^2)$ часу пішло на обробку.

Незважаючи на оптимальний час запиту, час попередньої обробки і особливо затрати пам’яті є завеликими (рис. 23).

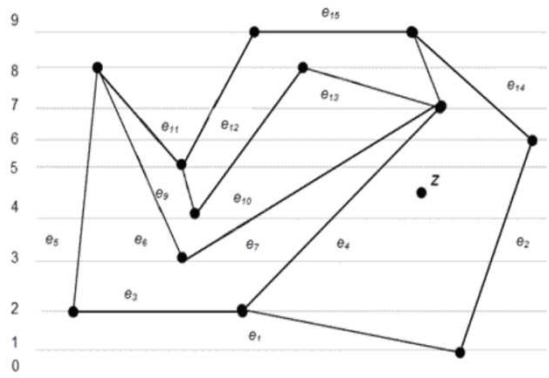


Рисунок 23. Локалізація точки в N-вершинному планарному розбиті методом плоского замітання

0. Статус: $\{\emptyset\}$. Вставка: $\{\emptyset\}$. Вилучення: $\{\emptyset\}$.

1. Статус: $\{e_1, e_2\}$. Вставка: $\{e_1, e_2\}$. Вилучення: $\{\emptyset\}$.

2. Статус: $\{e_5, e_3, e_4, e_2\}$. Вставка: $\{e_5, e_3, e_4\}$. Вилучення: $\{\emptyset\}$.

3. Статус: $\{e_5, e_6, e_7, e_4, e_2\}$. Вставка: $\{e_6, e_7\}$. Вилучення: $\{e_3\}$.

4. Статус: $\{e_5, e_6, e_9, e_{10}, e_7, e_4, e_2\}$. Вставка: $\{e_9, e_{10}\}$. Вилучення: $\{\emptyset\}$.

І так далі…

2.2.3 Метод ланцюгів

Ланцюг $C(u_1, \dots, u_p)$ – плоский прямолінійний граф з вершинами $\{u_1, \dots, u_p\}$ і ребрами $\{(u_i, u_{i+1}): i = 1, \dots, p-1\}$. Іншими словами, це *ламана лінія* (рис. 24).

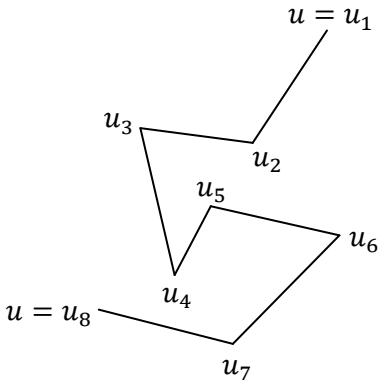


Рисунок 24. Ланцюг загального вигляду

Розглянемо планарне підрозбиття, яке визначається плоским прямолінійним графом G. Припустимо, що в G знайдений ланцюг C (підграф G) одного із типів:

1) C є циклом;
2) обидва кінці u_1 та u_p із C лежать на границі нескінченної області; в цьому випадку продовжимо C з обох кінців напівнескінченними паралельними протилежно напрямленими ребрами.

Ланцюг кожного типу ділить початкове підрозбиття на дві частини.

Дискримінацією точки z відносно ланцюга C називається процедура визначення того, по який бік від C лежить пробна точка z.

Ланцюг $C = (u_1, \dots, u_p)$ називається *монотонним відносно прямої* l, якщо будь-яка пряма, ортогональна до l, перетинає C тільки в одній точці. Тобто, будь-який монотонний ланцюг C належить до визначеного вище типу (2), а ортогональні проекції $(l(u_1), \dots, l(u_p))$ вершин C на l упорядковані вздовж l в порядку $l(u_1), \dots, l(u_p)$ (рис. 25).

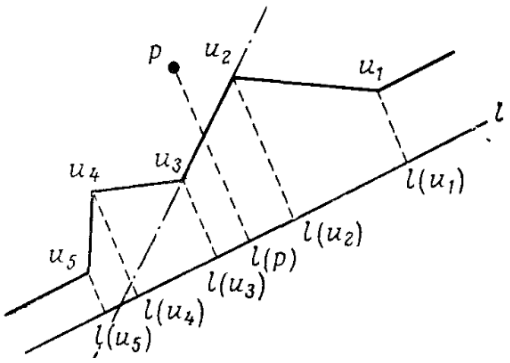


Рисунок 25. $L(u) = [l(u_1), l(u_2), l(u_3), l(u_4), l(u_5)]$,
$l(p)$ – проекція на l точки p та ланцюга.

Отже, процедура дискримінації точки p відносно ланцюга, монотонного відносно прямої l, зводиться до наступних двох кроків.

1. Локалізація $l(p)$ на $L(u)$ за $O(\log N)$ (отримуємо інтервал $(l(u_i), l(u_{i+1}+1))$.

2. Єдина перевірка, по який бік відносно ланцюга розташована точка p за $O(1)$.

Простий многокутник називається *монотонним*, якщо його границю можна розбити на два ланцюги, монотонних відносно однієї прямої.

Припустимо, що існує деяка множина $Z = \{C_1, \dots, C_r\}$ ланцюгів, монотонних відносно однієї і тієї ж прямої l, що мають такі властивості:

Властивість 1. Об'єднання всіх елементів Z містить заданий плоский прямолінійний граф G (при цьому конкретне ребро може належати більш ніж одному ланцюгу).

Властивість 2. Для будь-якої пари ланцюгів C_i і C_j із Z ті вузли C_i, які не є вузлами C_j, лежать по один бік від C_j (тобто ланцюги упорядковані).

Тоді множина Z називається *повною множиною монотонних ланцюгів графа* G.

Оскільки повна множина монотонних ланцюгів графа впорядкована, до неї можна застосувати двійковий пошук. При цьому в ролі елементарної операції порівняння виступає дискримінація точки відносно ланцюга.

Якщо є r ланцюгів в Z і в найдовшому ланцюзі p вершин, то пошук буде займати у найгіршому випадку $O(\log r * \log p)$ часу.

Побудувати повну множину монотонних ланцюгів для будь-якого плоского прямолінійного графа G неможливо. Однак плоский прямолінійний граф допускає побудову повної множини монотонних ланцюгів при певному обмеженні. Будь-який плоский прямолінійний граф можна легко перетворити в такий граф, до якого можна застосувати процедуру побудови ланцюгів.

Надалі вважатимемо прямою l вісь *y* на площині.

Нехай G – плоский прямолінійний граф з множиною вершин $\{v_1, \dots, v_N\}$, де вершини індексовані так, що $i < j \Leftrightarrow y(v_i) < y(v_j)$ або $y(v_i) = y(v_j)$ та $x(v_i) < x(v_j)$.

Вершина v_j називається *регулярною*, якщо існують такі цілі $i < j < k$, що (v_i, v_j) та (v_j, v_k) – ребра G. Говорять, що плоский прямолінійний граф G *регулярний*, якщо кожна вершина регулярна при $1 < j < N$ (крім двох крайніх вершин v_1 та v_N) (рис. 26).

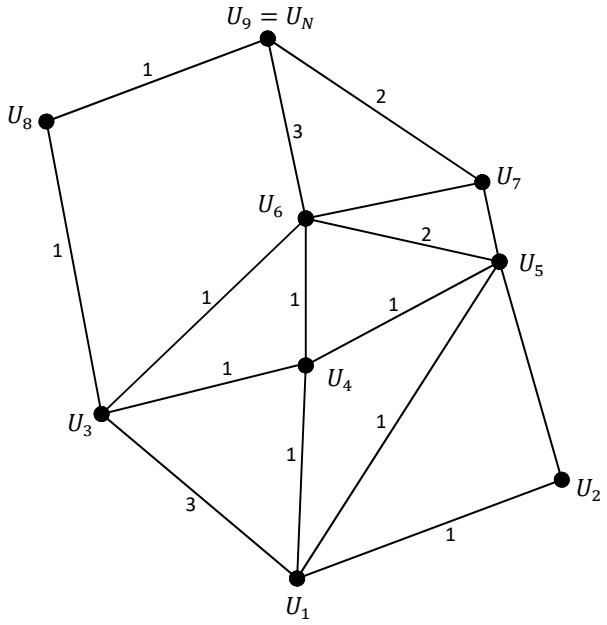


Рисунок 26. Плоский прямолінійний граф G

Вважатимемо, що ребро (v_i, v_j) орієнтоване від v_i до v_j, якщо $i < j$. Позначимо через $IN(v_j)$ та $OUT(v_j)$ множини ребер, які входять та виходять з вершини v_j. Нехай ребра в $IN(v_j)$ впорядковані за кутом проти годинникової стрілки, а ребра в $OUT(v_j)$ – за годинниковою стрілкою.

Теорема 5. Для довільної v_j $(j \neq 1)$ можна побудувати y-монотонний ланцюг від v_1 до v_j.

Доведемо за методом математичної індукції.

Для $j = 2$ – очевидно.

Нехай вірно для всіх $k < j$. Вершина v_j регулярна, тоді $\exists i < j$, що $(v_j, v_j) \in G$. Існує ланцюг C від v_1 до v_i, монотонний відносно осі y, і його з'єднання з ребром (v_j, v_j) дасть також монотонний ланцюг.

Теорема 6. Довільний регулярний граф можна розбити на повну множину ланцюгів, монотонних відносно осі y.

Позначимо через $W(e)$ вагу ребра e – кількість ланцюгів, яким належить e. Введемо позначення

$$W_{IN}(v) = \sum_{e \in IN(v)} W(e); W_{OUT}(v) = \sum_{e \in OUT(v)} W(e).$$

Ваги ребер обираються так, щоб виконувалися умови:

1) кожне ребро має додатну вагу;
2) для довільного v_j $(j \neq 1, j \neq N)$ $W_{IN}(v_j) = W_{OUT}(v_j)$.

Перша з умов гарантує, що кожне ребро належить хоча б одному ланцюгу.

Друга умова забезпечує, що через вершину v_j проходить $W_{IN}(v_j)$ ланцюгів, і їх можна вибрати так, щоб вони не перетиналися.

Теорема 7. Реалізація умови $W_{IN} = W_{OUT}$ є розв'язком потокової задачі і може бути досягнута за два проходи по графу G.

Покладемо спочатку $W(e) = 1$ для кожного ребра e. При першому проході від v_1 до v_N отримаємо $W_{IN}(v_j) \le W_{OUT}(v_j)$ для всіх некрайніх v_j. При другому проході від v_N до v_1 отримаємо $W_{IN}(v_j) \ge W_{OUT}(v_j)$. Отже, після двох проходів матимемо реалізацію умови $W_{IN}(v_j) = W_{OUT}(v_j)$.

Алгоритм балансування за вагою

Позначимо $W_{IN}(v) = |IN(v)|$, $W_{OUT}(v) = |OUT(v)|$.

procedure БАЛАНСУВАННЯ ЗА ВАГОЮ В РЕГУЛЯРНОМУ ППЛГ
  begin for кожного ребра e **do** $W(e) := 1$ (**ініціалізація**, див. рис. 27а)
  for $i := 2$ **to** $N - 1$ **do**
    begin $W_{IN}(v_i) :=$ сума ваг ребер, які входять в v_i ;
    $d_1 :=$ крайнє зліва ребро, яке виходить із v_i ;
    if $(W_{IN}(v_i) > v_{OUT}(v_i)$ **then** $W(d_1) := W_{IN}(v_i) - v_{OUT}(v_i) + 1$
    end (**перший прохід**, див. рис. 27б));
  for $i := N - 1$ **downto** 2 **do**
    begin $W_{OUT}(v_i) :=$ сума ваг ребер, які входять в v_i ;
    $d_2 :=$ крайнє зліва ребро, яке входить в v_i ;
    if $W_{OUT}(v_i) > W_{IN}(v_i)$ **then** $W(d_2) := W_{OUT}(v_i) - W_{IN}(v_i) + W(d_2)$
  end (**другий прохід**, див. рис. 27в))
end.

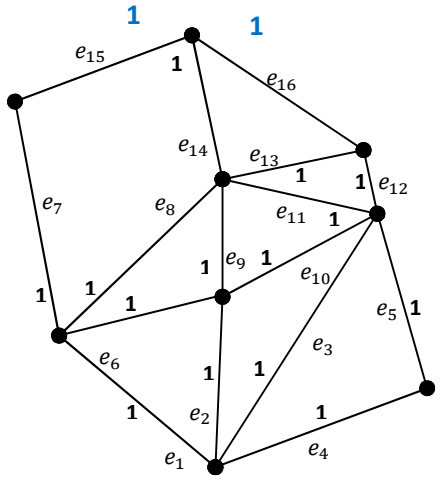


Рисунок 27а. Для кожного ребра $W(e) := 1$ (ініціалізація)

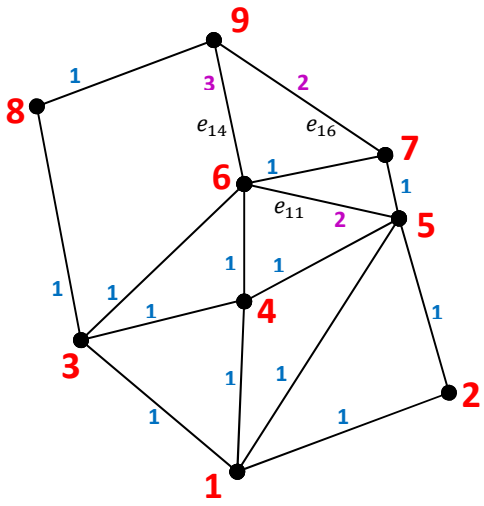


Рисунок 27б. Конфігурація ваг ребер після першого проходу:
(5): $(W_{IN}(5) = 3 > W_{OUT}(5) = 2) \Rightarrow W(e_{11}) := 3 - 2 + 1 = 2$;
(6): $(W_{IN}(6) = 4 > W_{OUT}(6) = 2) \Rightarrow W(e_{14}) := 4 - 2 + 1 = 3$;
(7): $(W_{IN}(7) = 2 > W_{OUT}(7) = 1) \Rightarrow W(e_{16}) := 2 - 1 + 1 = 2$.

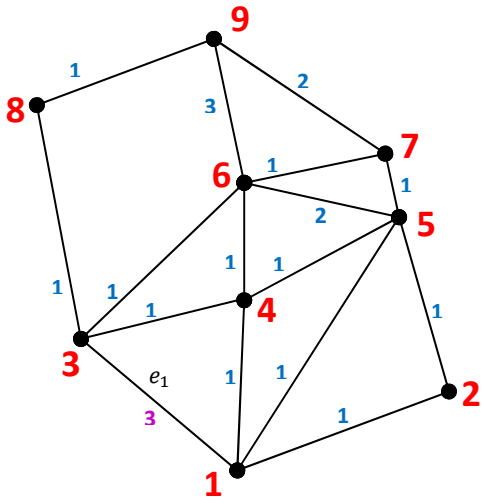


Рисунок 27в. Конфігурація ваг ребер після другого проходу:
(3): $(W_{OUT}(3) = 3 > W_{IN}(3) = 1) \Rightarrow W(e_1) := 3 - 1 + 1 = 3$

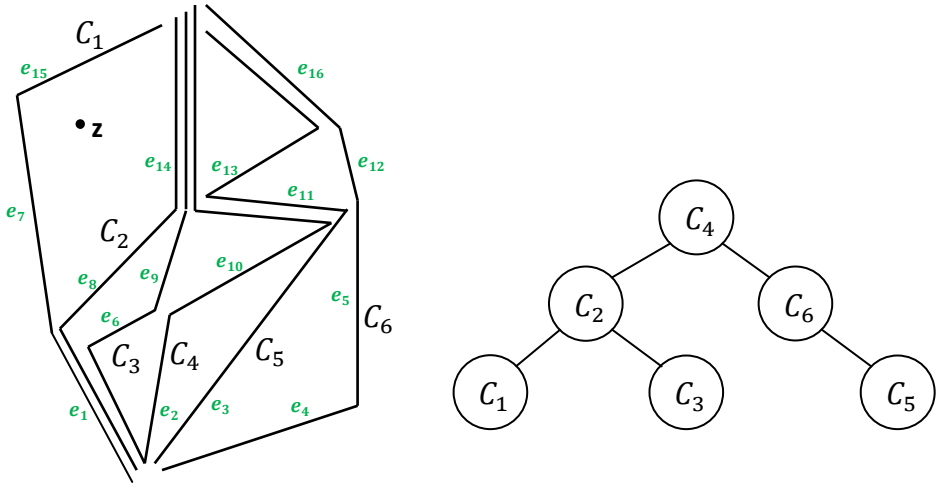


Рисунок 28. Повна множина монотонних ланцюгів та відповідне бінарне дерево

$C_1 = (e_1, e_7, e_{15})$, $C_2 = (e_1, e_8, e_{14})$,

$C_3 = (e_1, e_6, e_9, e_{14})$, $C_4 = (e_2, e_{10}, e_{11}, e_{14})$,

$C_5 = (e_3, e_{11}, e_{13}, e_{16})$, $C_6 = (e_4, e_5, e_{12}, e_{16})$.

$Z = (C_1, C_2, C_3, C_4, C_5, C_6)$ – повна множина монотонних ланцюгів.

Регуляризація графа

Нерегулярні вершини можна поділити на 2 типи (рис. 29):

- нерегулярні вершини першого роду (не мають вхідних ребер),
- нерегулярні вершини другого роду (не мають вихідних ребер).

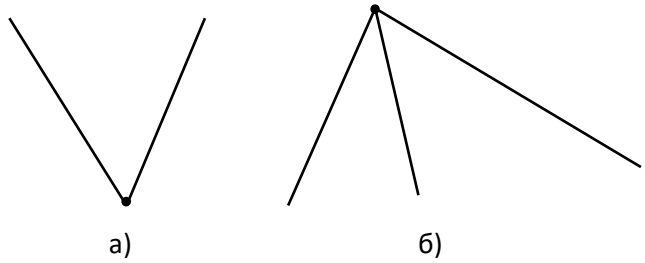


Рисунок 29. Типи нерегулярних вершин

Теорема 8. Довільний планарний граф можна перетворити в регулярний.

Використовуючи алгоритм плоского замітання, замітаємо граф згори донизу, регуляризуючи вершини, які не мають вихідних ребер, а потім знизу вгору для регуляризації вершин, які не мають вхідних ребер.

В процесі замітання для кожної вершини v реалізуємо наступні операції (рис. 30):

1) локалізуємо v (по абсцисі) в одному із інтервалів в структурі даних статусу;
2) коригуємо цю структуру статусу;
3) якщо *v* нерегулярна, тоді додаємо ребро від v до тієї вершини, яка зв'язана з інтервалом, який визначений в операції (1).

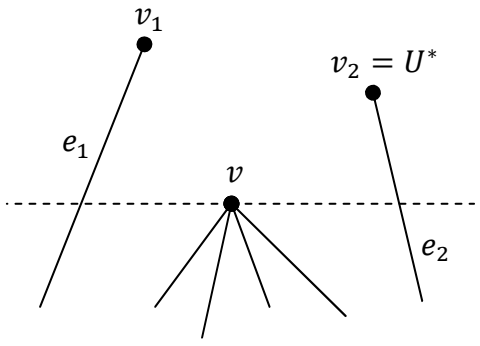


Рисунок 30. Процес замітання для вершини v

Уточнимо операцію (3). Очевидно, що для проведення ребер, що будуть регуляризовувати вершини необхідно знати дві вершини. Одна з яких – поточна нерегулярна, інша – та, що зустрічалася на шляху замітаючої прямої на попередньому кроці.

Основна проблема при введенні нового ребра полягає у проведенні його так, щоб воно не перетинало вже існуючі ребра. Ця умова дозволяє запропонувати такий вибір другої вершини ребра: вибирати таку вершину, яка лежить найближче (з півплощини, де замітаюча пряма вже була) до першої (нерегулярної) вершини між ребрами, що лежать зліва та справа від нерегулярної вершини (рис. 31).

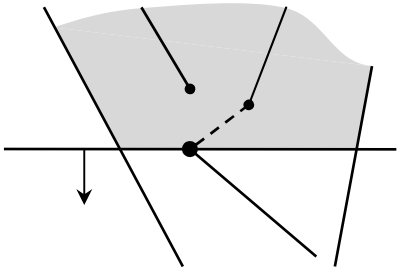


Рисунок 31. Введення нового ребра

Для замітаючої прямої встановлюється порядок ребер (для довільного набору ребер можна вказати таке $y = const$, що для будь-яких двох ребер можна сказати яке з них лівіше, а яке правіше; ребро вважається меншим за інше, якщо можна вказати таку точку на ньому, що горизонтальна пряма, яка проходить через цю точку, перетинає інше ребро в точці, що лежить правіше на прямій, ніж перша точка).

Статус замітаючої прямої буде містити не тільки ребра, що перетинаються прямою у поточній точці подій, а й найближчі точки між ребрами, що вже зустрічались як точки подій (див. табл. 2).

Найближча точка лівіше ребра 1	Ребро 1	Найближча точка між ребром 1 і ребром 2	Ребро 2	Найближча точка лівіше ребра 2

Таблиця 2. Пошук найближчих точок між ребрами

Таким чином, щоб скоригувати нерегулярну вершину, достатньо визначити, куди вона потрапляє у статусі та з'єднати її з точкою, найближчою у відповідному регіоні.

Теорема 9. N-вершинний плоский прямолінійний граф можна регуляризувати за час $O(N \log N)$ з витратою пам'яті $O(N)$.

Попередня обробка. Припустимо, що ППЛГ заданий у вигляді структури даних РСПЗ. Побудова множин $IN(v)$ і $OUT(v)$ – за $O(N)$. Процедура балансування за вагою – за $O(N)$. Регуляризація потребує $O(N \log N)$ часу, тому час попередньої обробки займає $O(N \log N)$.

Теорема 10. Локалізацію точки в N-вершинному планарному підрозбитті можна реалізувати методом ланцюгів за час $O(\log_2 N)$ з використанням $O(N)$ пам'яті при витратах $O(N \log N)$ часу на попередню обробку.

На рис. 32 приклад регуляризованого плоского прямолінійного графа. Штриховані ребра отримані при замітанні зверху вниз, пунктирні – при замітанні знизу нагору.

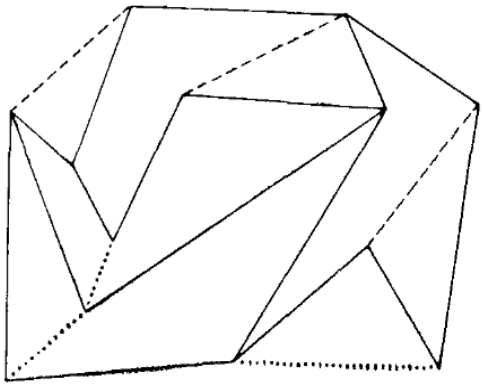


Рисунок 32. Приклад регуляризованого плоского прямолінійного графа

2.2.4 Метод деталізації триангуляції (Кіркпатрік)

Триангуляція на множині вершин V на площині – плоский прямолінійний граф з максимальним числом ребер, що не перевищує $(3|V| - 6)$ (за формулою Ейлера).

Нехай задана N-вершинна триангуляція G. Вважатимемо, що будується послідовність триангуляцій $T_1, T_2, \dots, T_{h(N)}$, де $T_1 = G$, а T_i отримується з T_{i-1} за такими правилами:

Крок (1). Вилучимо деяку максимальну множину несуміжних неграничних вершин T_{i-1} та інцендентні їм ребра.

Крок (2). Триангулюємо многокутники, які утворились в результаті вилучення вершин та ребер.

Таким чином $T_{h(N)}$ не має внутрішніх вершин. Усі триангуляції $T_1, T_2, \dots, T_{h(N)}$ мають одну загальну границю, оскільки на кроці (1) ми вилучали лише внутрішні вершини.

Трикутники будемо позначати літерою R з індексами. Трикутник R_j може з'явитися в багатьох триангуляціях, ми вважатимемо, що R_j належить триангуляції T_i (позначимо $R_j \in^* T_i$), якщо R_j створений на кроці (2) при побудові T_i.

Структура K, топологію якої визначає направлений ациклічний граф, визначається наступним чином: від трикутника R_k до трикутника R_j проводиться дуга, якщо при побудові T_i після T_{i-1} ми маємо:

- R_j вилучається з T_{i-1} на кроці (1);
- R_k створюється в T_i на кроці (2);
- $R_j \cap R_k = \emptyset$.

Критерій вибору вершин.

Локалізація точки заданим методом буде найшвидшою у випадку пошукового дерева найменшої висоти. Щоб досягти цього, необхідно на кожному кроці вилучати максимально можливу множину несуміжних вершин із максимальним ступенем.

Для великої кількості точок можна запропонувати наступний підхід щодо вибору та вилучення множини точок: починаючи з довільної вершини, помічаємо її сусідів (які не можуть вилучатися на поточному кроці) і продовжуємо доти, поки ще є непомічені сусіди.

Вважатимемо, що можна вибрати множину так, щоб виконувались наступні властивості (через N_i позначено число вершин в T_i):

Властивість 1. $N_i = \alpha_i N_{i-1}$, де $\alpha_i \le \alpha \le 1$ для i $= 2, \ldots, h(N)$.

Властивість 2. Кожний трикутник $R_j \in T_i$ перетинається не більше ніж з H трикутниками з T_{i-1}, і навпаки.

З властивості 1 як наслідок випливає, що $h(N) \le \left[\log_{\frac{1}{\alpha}} N\right] = O(\log N)$, оскільки при переході від T_{i-1} до T_i вилучається фіксована доля вершин.

Із властивостей 1 та 2 випливає, що пам'ять для K рівна $O(N)$. Ця пам'ять використовується для збереження вузлів і вказівників на їх нащадків.

З теореми Ейлера про плоскі графи випливає, що T_i містить $F_i < 2N_i$ трикутників. Число вузлів K, які представляють трикутники із T_i, не перевищує F_i. Звідси випливає, що загальна кількість вузлів K менша, ніж

$$2\left(N_1 + N_2 + \cdots + N_{h(N)}\right) \le 2N_1\left(1 + \alpha + \alpha^2 + \cdots + \alpha^{h(N)-1}\right) < 2N/(1-\alpha)$$

Пам'ять для вказівників: за властивістю 2 кожен вузол має $\le H$ вказівників, звідки $\le 2NH/(1-\alpha)$ вказівників з'явиться в K.

Теорема 11. Локалізацію точки в N-вершинному планарному підрозбитті методом деталізації триангуляції можна зробити за час $O(\log N)$, з використанням $O(N)$, пам'яті, якщо $O(N \log N)$, часу пішло на попередню обробку.

Це перший метод з розроблених, що поєднував логарифмічний час запиту з лінійними витратами пам’яті. Однак значення деяких констант в цих оцінках є високими.

Граф повинен мати трикутну зовнішню границю. Для цього на початку ми оточуємо його великим трикутником (рис. 33).

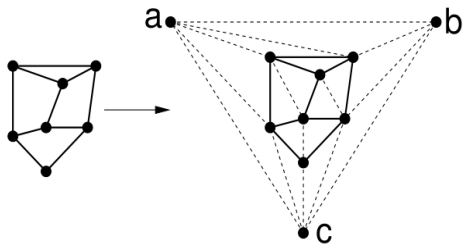


Рисунок 33. Оточення графу трикутною зовнішньою границею

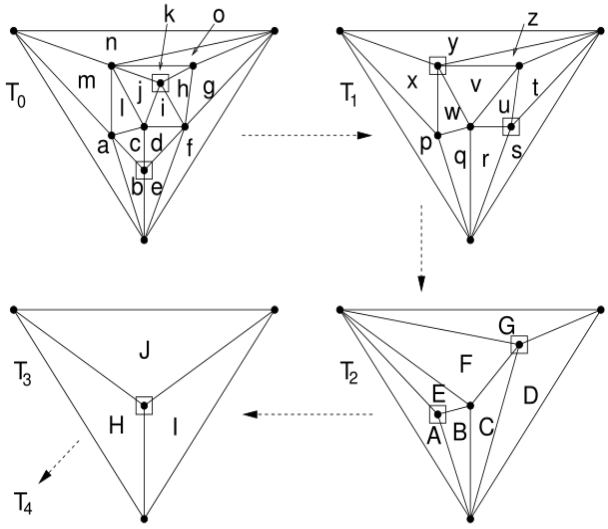


Рисунок 34. Приклад послідовності триангуляцій. Слід зауважити, що тут після кожної триангуляції всі трикутники заново перейменовуються, навіть ті, які не змінилися

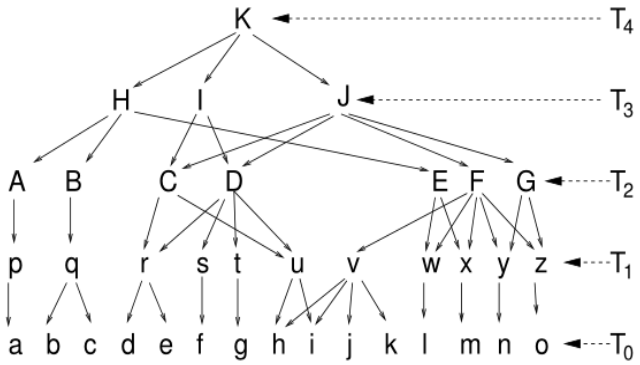


Рисунок 35. Направлений граф пошуку. Зверніть увагу на ідентичність, зокрема, трикутників A, p та a чи x та m.

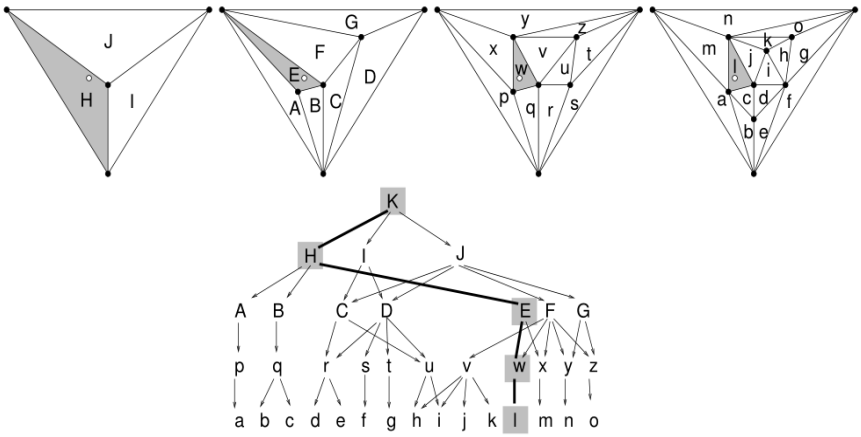


Рисунок 36. Приклад локалізації точки

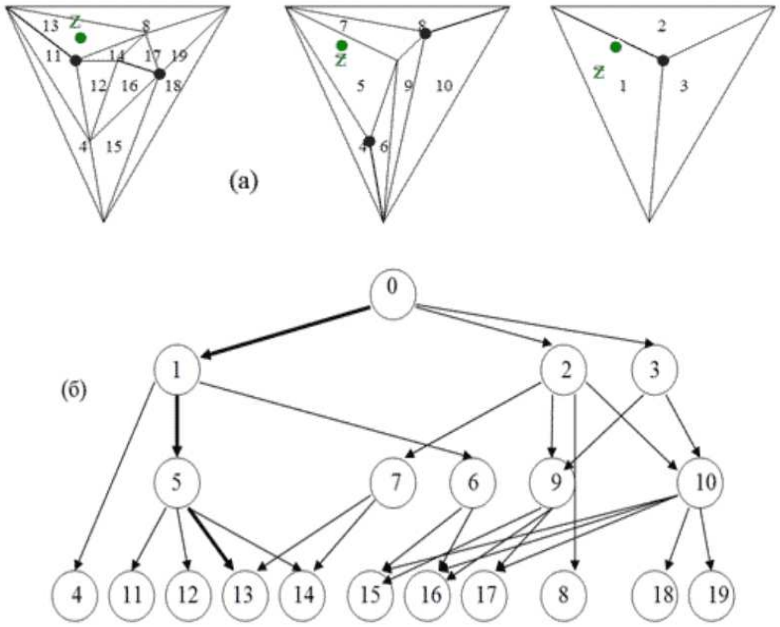


Рисунок 37. Послідовність триангуляцій (а)
та відповідний направлений граф пошуку (б)

2.2.5 Метод трапецій

Метод можна вважати розвитком методу смуг, причому тепер наявність у графі довгих майже вертикальних ребер перетворюється з недоліку в перевагу.

Трапеція має два горизонтальні боки та може мати дві, одну чи нуль бокових сторін, причому наявні бокові сторони є ребрами плоского прямолінійного графа чи їх частинами, і жодне інше ребро ППЛГ не перетинає її обидва горизонтальні боки.

Пошук здійснюється шляхом локалізації пробної точки в послідовності вкладених трапецій, поки не отримаємо трапецію, усередині якої немає ребер графа чи їх фрагментів.

Множина вершин V графа G впорядкована за зростанням їх оридинат, множина ребер E упорядкована згідно відношення часткового порядку $<$: для двох ребер e_1 та e_2 запис

$e_1 < e_2$ означає, що існує горизонталь, яка перетинає обидва ребра, і точка її перетину з e_1 знаходиться лівіше за точку перетину з e_2.

Механізм побудови структури даних пошуку обробляє по одній трапеції та намагається розбити її на максимально можливу кількість менших трапецій. Це відбувається шляхом розрізання трапеції R на нижню R_1 та верхню R_2 горизонтальною прямою, яка є медіаною множини ординат вершин всередині R (рис. 38).

Якщо ребро графа G перетинає обидві горизонтальні сторони трапеції, то воно називається *закриваючим*.

Після визначення медіани y_{med} трапеції R множина ребер графа G, яка перетинає R, проглядається зліва направо та розділяється на дві множини, що відносяться до R_1 та R_2. Якщо зустрічається закриваюче ребро, воно стає правою боковою границею нової трапеції, яка обробляється незалежно (рис. 38).

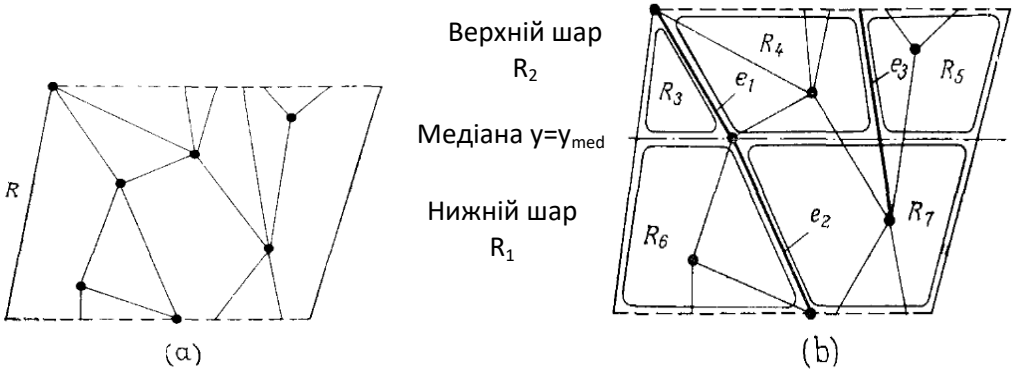


Рисунок 38. Трапеція R (a), визначення медіани y_{med} (b)

Розбиття трапеції

Кожній трапеції R відповідає дерево бінарного пошуку $T(R)$ (рис. 39), з лінійною перевіркою кожного вузла, які можуть бути двох типів:

- ∇-вузли, що відповідають перевірці по горизонталі,
- O-вузли, що відповідають перевірці відносно прямої, яка містить ребро графа.

Корінь завжди є ∇-вузлом. Кількість ∇-вузлів у дереві пошуку дорівнює $N-2$ (крайні вершини не беруть участі в розбитті).

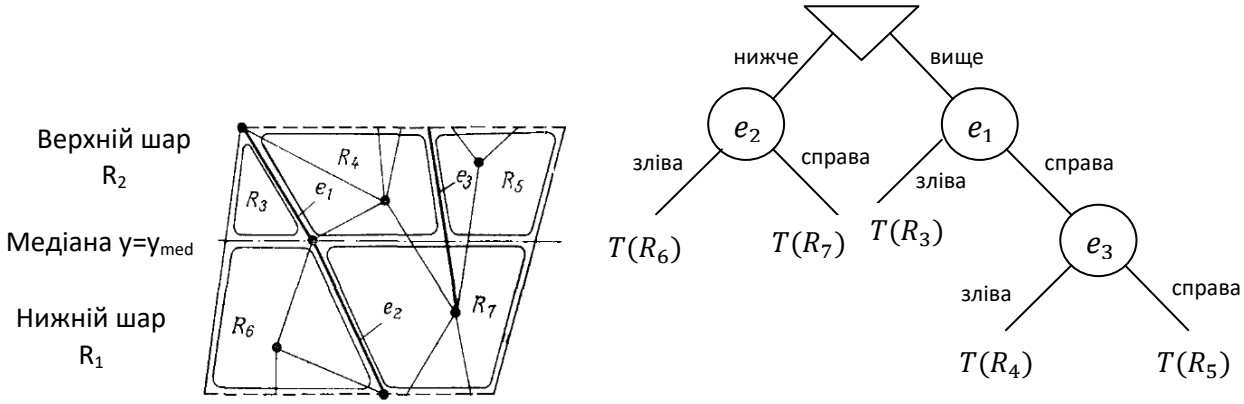


Рисунок 39. Розбиття трапеції R та відповідне дерево бінарного пошуку $T(R)$

В алгоритмі виділяються три основні дії:

1) визначення медіани множини ординат вершин в R;
2) розбиття нижнього і верхнього ярусів на трапеції та отримання для кожного яруса $R_i(i = 1,2)$ ланцюга U_i, який складається з ребер та дерев;
3) балансування двох ланцюгів U_1 та U_2.

Процедура БАЛАНС потребує $O(h \log h)$ часу: розділення U – $O(h)$ часу, два рекурсивних виклики процедури, які застосовуються до половин вихідного ланцюга.

Аналіз висоти збалансованих дерев: $h < N$, де h – кількість дерев в ланцюзі U (і нових трапецій), N – число вершин графа.

Якщо многокутник має s вершин, то в ньому $\leq s - 3$ діагоналей. Оскільки $s \leq N$, то $h \leq s - 2 \leq N - 2$.

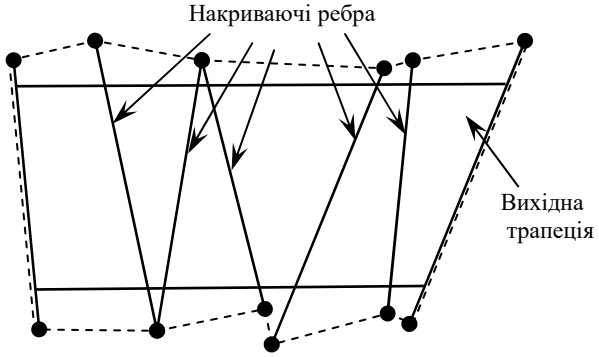


Рисунок 40.

Теорема 12. Для заданого ланцюга $U = T_1 e_1 \dots e_{h-1} T_h$ висота $d(U)$ дерева, побудованого за допомогою процедури БАЛАНС, не перевищує $3 \log_2 W(U) + [\log_2 N] + 3$.

Для плоского прямолінійного графа справедлива нерівність $W(U) \leq N$, отже висота дерева пошуку для графа не перевищує $4[\log_2 N] + 3$

Тут $W(U)$ – кількість вершин графа, що міститься в ланцюзі (дереві) U.

Теорема 13. Точку можна локалізувати в N-вершинному планарному підрозбитті методом трапецій менше ніж за $4\log_2 N$ перевірок, з використанням $O(N\log N)$ пам'яті і такого ж часу попередньої обробки.

```
function ТРАПЕЦІЯ(E, V, I)
    begin
        if (V ≠ ∅) then return L                    (*листок дерева пошуку*)
        else begin
            E1 := E2 := V1 := V2 := U := ∅;
            y_med := медіана множини ординат V;
            l1 := [min(l), y_med]; l2 := [y_med, max(l)];
            repeat e ⇐ E
                for i = 1,2 do begin
                    if (e має кінец p в середині R_i) then begin
                        E_i ⇐ e; V_i := V_i ∪ {p}
                    end;
                    if e накриває R_i then begin
                        U_i ⇐ ТРАПЕЦІЯ(E_i, V_i, I_i);
                        if (e ≠ L) then U_i ⇐ e;
                        E_i := V_i := ∅
                    end
                end
            until e = L;
            новий(w);                               (*створення нового вузла w, кореня T(R)*)
            Y[w] := y_med;                          (*дискримінація вузла w*)
            ЛДЕРЕВО[w] := БАЛАНС(U1);               (*функція БАЛАНС приймає на вході
                                                    послідовність із дерев та ребер і
                                                    організовує їх в збалансоване дерево*)

            ПДЕРЕВО[w] := БАЛАНС(U2);
            return ДЕРЕВО[w]
    end
end.
```

Медіану можна шукати іншим способом. Вершини трапеції розташовуються в масиві за зростанням ординат, і його медіана знаходиться за одне обернення. Модифікація процедури ТРАПЕЦІЯ наступна. Послідовність ребер проглядається двічі. При першому проході кожна вершина просто помічається іменем тієї трапеції, до якої вона належить, будується масив вершин для кожної породженої трапеції. При другому проході виконується цикл **repeat**.

2.3 Задачі регіонального пошуку

2.3.1 Особливості постановки. Одновимірний регіональний пошук

Задачі регіонального пошуку можна розглядати в певному смислі як двоїсті до задач локалізації точки.

«Файлом» вважаємо набір структур, кожна з яких ідентифікується впорядкованим d-плексом ключів $(x_1, \dots, x_d)$. Кожен такий d-плекс можна розглядати як точку d-вимірного декартового простору.

Запит у цій моделі визначає область (регіон) у d-вимірному просторі, а результатом пошуку може бути

– звіт про множину точок файлу, яка міститься у цій області (*запит у режимі звіту*);

– підрахунок числа точок в області запиту (*запит в режимі підрахунку*).

Подібні задачі часто виникають у географії, статистиці, в системах автоматизованого проектування та при роботі з базами даних (рис. 41, 42).

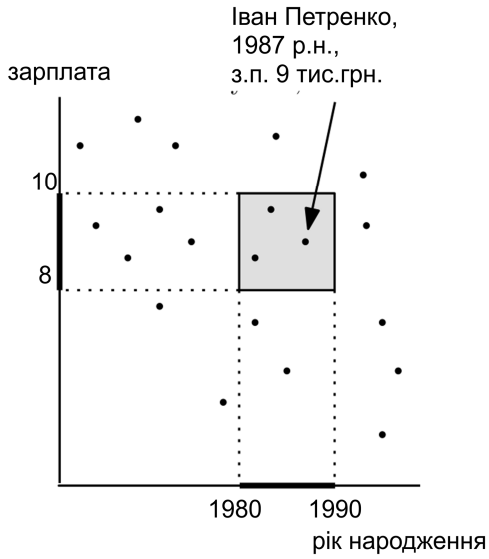


Рисунок 41. Приклад запиту до бази даних: знайти кількість робітників віком від 30 до 40 років із заробітною платою від 8 до 10 тис. грн. («Пошук по багатьом ключам»)

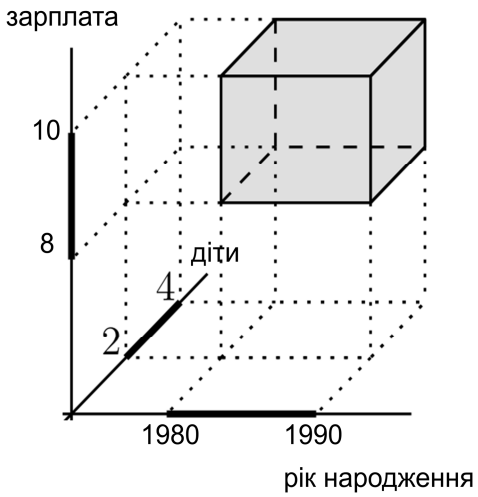


Рисунок 42. Приклад 3-вимірного регіону запиту: рік народження з 1980 по 1990, заробітна плата від 8 до 10 тис. грн., від 2 до 4 дітей.

Задачі регіонального пошуку можна класифікувати за типом області пошуку:

- ортогональний пошук («прямокутний» регіон);
- пошук в симплексі («трикутний» регіон);
- пошук в півплощині;
- пошук в диску («круговий» регіон).

Кожен з цих типів регіонального пошуку вимагає своїх окремих принципів і підходів до розв'язання задач.

Ми розглянемо задачу ортогонального пошуку.

Розглядатимемо лише масові запити. Збудовані структури даних будуть статичними (без модифікацій після побудови). За таких умов при оцінці ефективності методів достатньо зосередитись на часі пошуку і витратах пам'яті (передобробка одноразова та часто має порядок оцінки пам'яті).

Припустимо, що «файл» – це фіксований набір S із N записів. Відповіддю на запит буде множина $S' \subset S$.

Відповідь на запит у режимі підрахунку – єдине ціле число k (k – потужність S').

Відповідь на запит в режимі звіту – імена усіх k елементів S'.

Можна виділити два типи дій при обробці запиту:

1) *пошук*, тобто дії, які приводять до елементів шуканої підмножини (зазвичай, послідовність порівнянь);
2) *вибірка*, тобто дії по складанню відповіді на запит (для запитів у режимі звіту – це фактичне визначення шуканої підмножини).

При аналізі запитів у режимі підрахунку всю обчислювальну роботу вбирає у себе «пошук»; при цьому очікується, що час запиту для гіршого випадку буде функцією $f(N,d)$, яка залежить від розміру файлу та розмірності простору.

При аналізі в режимі звіту обчислювальна робота є комбінацією пошуку і вибірки, і та її частина, яка пов'язана з вибіркою, обмежена знизу числом k – потужністю S'.

Нехай k – потужність вибірки S'. У загальному випадку верхня оцінка часу запиту у режимі звіту: $O(f(N,d)+k\cdot g(N,d))$, де $f(N,d)$ – «час пошуку» та $k\cdot g(N,d)$ – «час вибірки».

$\Omega(k)$ – тривіальна нижня оцінка часу вибірки.

$\Omega(\log Q(S))$ – нижня оцінка кількості порівнянь для пошуку (використовується модель двійкового дерева розв'язків, яка підраховує число порівнянь, необхідних для доступу до елементів множини $S'\subset S$ всередині регіону запиту), де $Q(S)$ – число різних підмножин S, отриманих як відповіді на запити.

Нехай S – множина усіх точок, які мають одну і лише одну ненульову цілочисельну координату в інтервалі $[-a,a]$ (рис. 43).

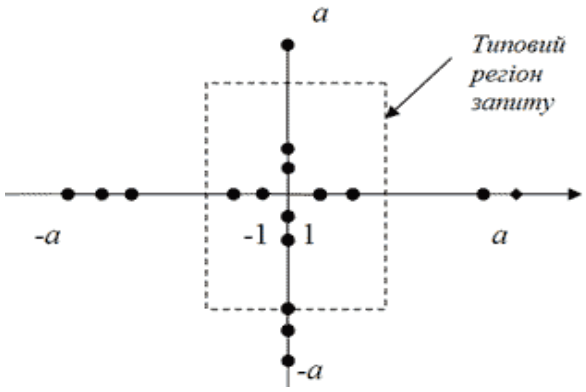


Рисунок 43. Приклад регіону запиту

Тоді $N=2ad$ (тут $d=2$).

Візьмемо $x\in[-a,-1]$ та $y\in[1,a]$. Вони визначать регіон запиту, для якого підмножина пов’язаних з ним точок непорожня й відмінна від усіх інших підмножин.

Цей вибір можна здійснити $a^{ad}=(N/(2d))^{2d}$ способами, тому нижня оцінка кількості двійкових розв'язків буде $\Omega(\log(N/(2d))^{2d})=(d\log N)$.

$\Omega(Nd)$ – тривіальна нижня оцінка пам'яті, яка зайнята під структуру даних пошуку.

Для всіх алгоритмів, описаних далі, час пошуку $O(f(N,d)+k)$.

Одновимірний регіональний пошук

В цьому випадку «файл» – множина із N точок на осі x, запитний регіон – відрізок $[x', x'']$ (називається x-регіоном).

Ефективний регіональний пошук реалізується способом, який базується на методі двійкового пошуку: лівий кінець x' x-регіона локалізується на осі x дихотомією, потім рухаємось вздовж цієї осі за зростанням, поки не досягнемо правого кінця x''.

Структура даних, яка забезпечує вказану дію, є збалансованим прошитим бінарним деревом пошуку (його листки додатково зв'язані у списку, який виражає порядок абсцис). Дерево і список обробляються відповідно на фазах пошуку і вибірки.

Оцінки: оптимальна як за часом запиту $\Omega(\log N + k)$, так і за пам'яттю $\Omega(N)$.

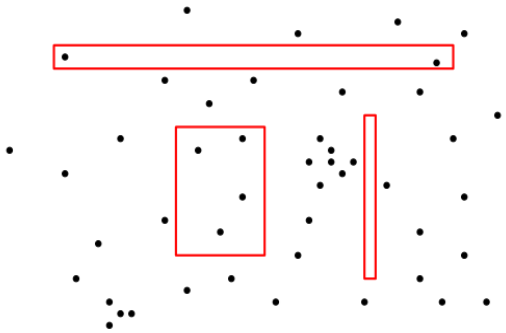


Рисунок 44. Приклад двовимірного регіонального пошуку

2.3.2 Метод багатовимірного двійкового дерева (Kd-дерева)

Розглянемо двовимірний випадок ($2d$-дерево). Метод можна вважати узагальненням дихотомії.

Загальна ідея:

1 припустимо, що вся площина є нескінченним прямокутником, який будемо «розрізати» на частини;

2 почергово розбиватимемо множину точок по x-координаті та y-координаті;

3 по x-координаті: проводиться вертикальна лінія так, щоб по обидва боки знаходилася (приблизно) однакова кількість точок множини;

4 по y-координаті: проводиться горизонтальна лінія так, щоб зверху і знизу була (приблизно) однакова кількість точок.

(Узагальнений) прямокутник: область на площині, яка визначена декартовим добутком $[x_1, x_2] \times [y_1, y_2]$, включно з граничними випадками, коли в будь-якій комбінації допускається $x_1 = -\infty, x_2 = \infty,\ y_1 = -\infty,\ y_2 = -\infty$.

Процес розбиття S шляхом розрізання площини краще за все ілюструвати в поєднанні з побудовою двовимірного двійкового дерева T: з кожним вузлом v неявно зв'язуються прямокутник $\boldsymbol{R}(v)$ та підмножина точок $S(v) \subset S$ точок, що лежать всередині $\boldsymbol{R}(v)$.

Явно з v буде пов'язана точка $P(v) \in S(v)$ і розрізаюча пряма $l(v)$, що проходить вертикально чи горизонтально через $P(v)$. Процес розбиття завершиться, коли з'явиться прямокутник, який не містить всередині жодної точки, відповідний йому вузол є листком дерева T (рис. 45).

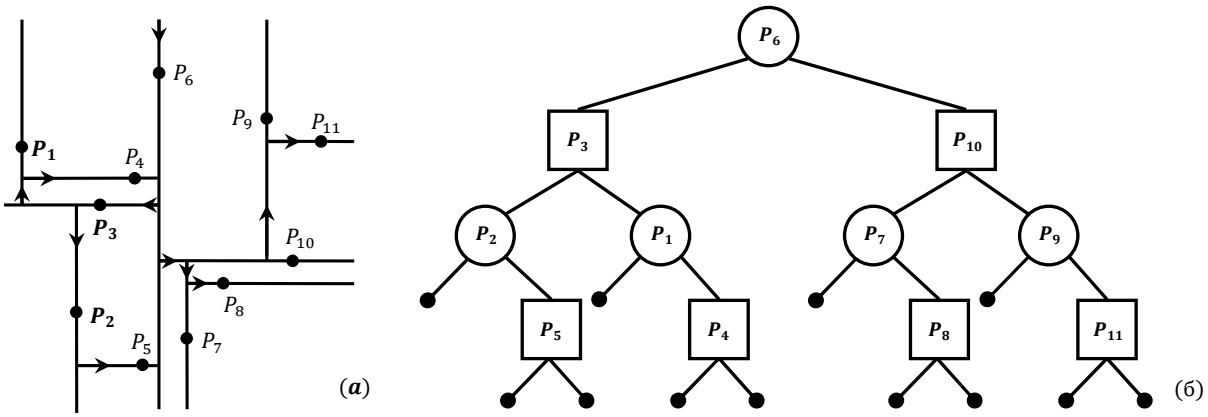


Рисунок 45. Процес розбиття S шляхом розрізання площини (а) та відповідне двовимірне двійкове дерево T (б)

Дерево міститиме три типи вузлів:

- кружки – нелистові вузли з вертикальною лінією розрізу,
- квадрати – нелистові вузли з горизонтальною лінією розрізу,
- точки – листки.

Пошук в $2d$-дереві відбувається за схемою «розділяй та владарюй».

Кожен вузол v містить параметри: P(v) – поточна точка, $\boldsymbol{R}(v)$ – прямокутна область, $S(v)$ – множина точок, що належить $\boldsymbol{R}(v)$, $l(v)$ – розрізаюча пряма, D – регіон запиту. Від взаємного розташування $\boldsymbol{R}(v)$, $S(v)$, $l(v)$ та D для кожного вузла $v \in$ T залежить регіональний пошук.

Для кожної v пряма $l(v)$ розбиває $\boldsymbol{R}(v) = \boldsymbol{R}_1(v) \cup \boldsymbol{R}_2(v)$. Перевіряється $D \cap \boldsymbol{R}_1(v)$, $D \cap \boldsymbol{R}_2(v)$.

1. $D \cap \boldsymbol{R}_1(v) \neq \emptyset$ та $D \cap \boldsymbol{R}_2(v) = \emptyset \Rightarrow$ лівий пошук (в $D \cap \boldsymbol{R}_1(v)$).
2. $D \cap \boldsymbol{R}_1(v) = \emptyset$ та $D \cap \boldsymbol{R}_2(v) \neq \emptyset \Rightarrow$ правий пошук (в $D \cap \boldsymbol{R}_2(v)$).).

3. $D \cap \boldsymbol{R}_1(v) \neq \emptyset$ та $D \cap \boldsymbol{R}_2(v) \neq \emptyset \Rightarrow$ перевірка «$P(v) \in D$?», потім лівий і правий пошуки (в $D \cap \boldsymbol{R}_1(v)$ та $D \cap \boldsymbol{R}_2(v)$ відповідно).

Процес пошуку завершується при досягненні листа. У тих вузлах, де пошук розгалужується, відбувається перевірка «$P(v) \in D$?». Вибрані вузли помічені зірочкою $*$: множина $\{p_3, p_4, p_8\}$ (рис. 46).

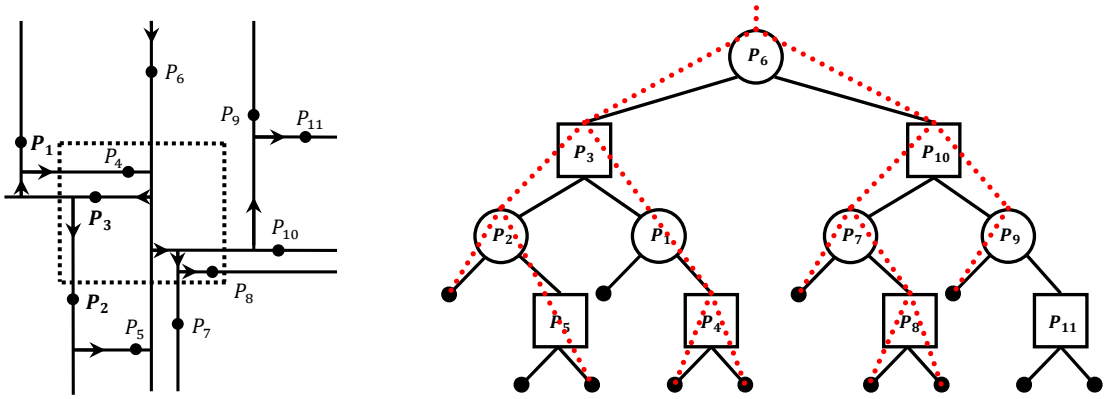


Рисунок 46. Ілюстрація метода пошуку за допомогою двовимірного двійкового дерева

Строго: $v \in \mathrm{T}$ характеризується трійкою $(\mathrm{P}(v), t(v), \mathrm{M}(v))$. Тут пара $(t(v), M(v))$ визначає пряму $l(v)$: $t(v)$ визначає її горизонтальність $(y = M(v) = y(P(v)))$ чи вертикальність $(x = M(v) = x(P(v)))$ $l(v)$.

Алгоритм накопичує точки у списку U – зовнішньому до процедури і спочатку порожньому. Позначимо через $D = [x_1, x_2] \times [y_1, y_2]$ регіон запиту.

```
procedure ПОШУК(v, D);
  begin
      if (t(v) = вертикаль) then
          [l, r]:= [x1, x2]
      else [l, r]:= [y1, y2];
      if (l ≤M(v) ≤r) then
         if (P(v) ∈D) then
                  U ⇐ P(v);
      if (v ≠листок) then
         begin
             if (l < M(v)) then ПОШУК(ЛСИН[v], D);
             if (M(v) < l) then ПОШУК(ПСИН[v], D);
         end;
  end.
```

Оцінимо складність. Пам'ять – $\Theta(N)$ (по вузлу на точку). Побудова дерева – $\Theta(N \log N)$ наступним способом.

Розріз множини S проводиться в результаті обчислення медіани множини x-координат (y-координат) точок з S за час $O(|S|)$, і шляхом формування розбиття S з такою ж оцінкою часу.

За час $O(N)$ вихідна множина розбивається, в результаті чого отримуємо півплощини, в кожній із яких по $N/2$ точок. Отримуємо рекурентне співвідношення для часу $T(N)$ роботи алгоритму побудови дерева: $T(N) \leq 2T(N/2) + O(N)$.

Можна також попередньо відсортувати точки по x- та y-координатах і робити розбиття на цій основі за $O(N)$.

Аналіз часу запиту для найгіршого випадку.

- Якщо час витрачений у вузлі v не даремно, то точка $P(v)$ вибирається (продуктивний вузол); інакше цей вузол є непродуктивним.
- Ситуація найгіршого випадку: побудувати максимальне піддерево відвіданих вузлів, всі з яких непродуктивні.
- Перетин запитного регіону D з $\boldsymbol{R}(v)$ можна віднести до різних «типів» в залежності від кількості сторін $\boldsymbol{R}(v)$, які мають непорожні перетини з D (рис. 47).

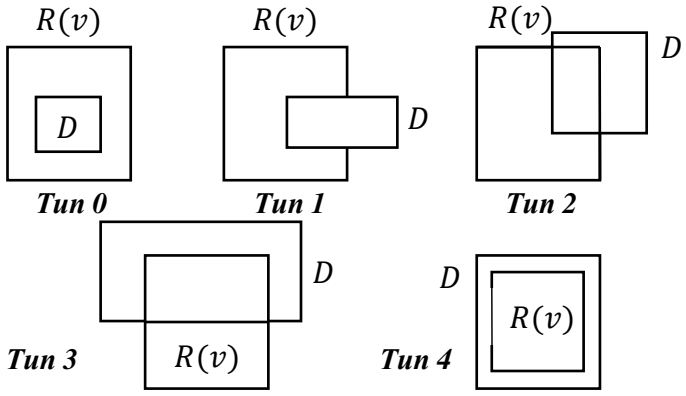


Рисунок 47. Типи перетину запитного регіону D з $\boldsymbol{R}(v)$

Єдиний тип, який завжди є продуктивним – тип 4.

Ситуація, коли перетин типу 2 на висоті m непродуктивний і породжує перетин типу 2 і перетин типу 3 на висоті $(m-1)$ (обидва можна зробити непродуктивними) (рис. 48а).

Ситуація, коли перетин типу 3 на висоті m непродуктивний і породжує пару перетинів типу 3 на висоті $(m-2)$ (їх можна зробити непродуктивними) (рис. 48б).

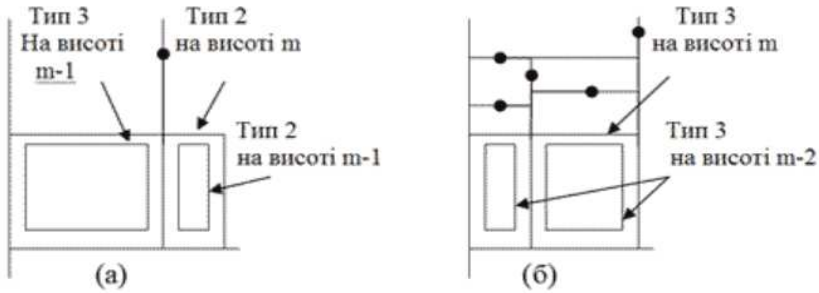


Рисунок 48.

Позначимо через $U_i(m)$ число пройдених непродуктивних вузлів в піддереві висотою m, корінь якого має тип i ($i = 2,3$). Одержимо рекурентні співвідношення:

$$U_2(m) = U_2(m-1) + U_3(m-1) + 1$$

$$U_3(m) = 2U_3(m-2) + 3$$

Теорема 14. Для файлу потужністю N в найгіршому випадку час запиту становить $O\sqrt{N}$, навіть якщо вибрана множина виявиться порожньою.

Розглянемо багатовимірний випадок. Результуюче розбиття d-вимірного простору моделюється двійковим деревом з N вузлами, яке називається багатовимірним двійковим деревом. Наприклад, для 3 вимірів регіоном запиту буде прямокутний паралелепіпед, і в процесі побудови дерева будуть задіяні розрізаючі площини, паралельні осям координат.

Аналіз ефективності можна узагальнити на випадок d вимірів:

Теорема 15. За допомогою методу Kd-дерева регіональний пошук на d-вимірній множині (при $d \geq 2$) з N точок можна провести за час $O(d \cdot N^{1-1/d} + k)$, де k – кількість знайдених точок, з використанням $\Theta(dN)$ пам'яті при витраті $\Theta(dN \log N)$ часу на попередню обробку.

2.3.3 Метод дерева регіонів

Метод Kd-дерева дає неефективний результат для найгіршого випадку, тому шукалися інші методи з кращою оцінкою часу пошуку.

Розглянемо множину на осі x, яка складається з N абсцис, які нормалізовані до цілих з інтервалу $[1, N]$ по всій величині. N абсцис визначають $N-1$ елементарний відрізок $[i,\ i+1]$ для i $= 1, 2, \ldots, N-1$.

Будь-який відрізок, кінці якого належать множині із N заданих абсцис, може бути розбитий деревом відрізків $T(1, N)$ на максимум $2[\log_2 N] - 2$ стандартних відрізки. Кожен стандартний відрізок віднесений до одного з вузлів $T(1, N)$, а ті вузли, які визначають розбиття відрізка $[i, j]$ – це вузли віднесення.

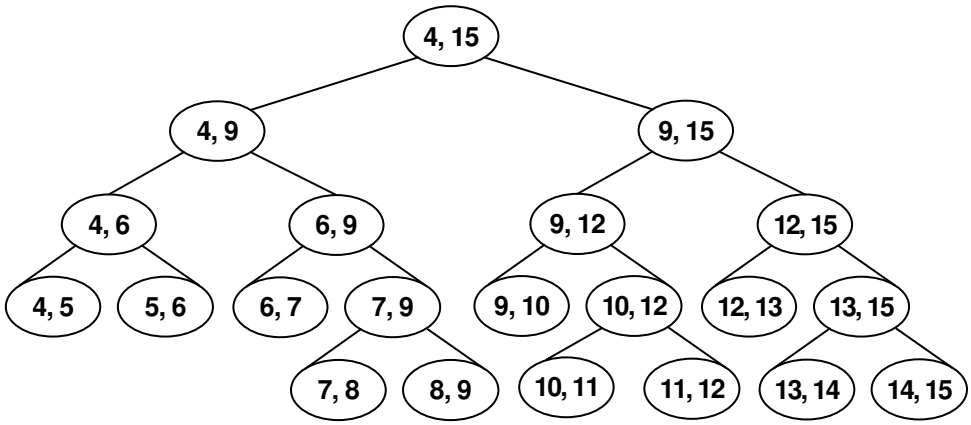


Рисунок 49. Приклад дерева відрізків $T(4,15)$

Дерево відрізків $T(1,N)$ використовується при пошуку по x-координаті. Цей пошук визначає універсальну множину вузлів (вузлів віднесення). Кожен такий вузол v відповідає підмножині точок із абсцисами, що належать даному відрізку. Ординати цих точок утворюють звичайне двійкове дерево для регіонального пошуку в y-напрямку.

Так побудована структура даних – *дерево регіонів* (рис. 50):

- його первинною структурою є дерево відрізків, заданих на абсцисах точок вихідної множини S;
- в кожному вузлі цього дерева є вказівник на прошите двійкове дерево пошуку (вторинні структури).

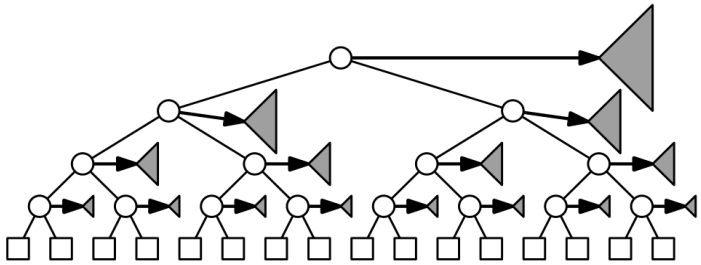


Рисунок 50. Структура дерева регіонів

Проекції на вісь OX заданої множини точок розбивають вісь абсцис на множину інтервалів, на основі чого будується дерево відрізків. Вузлами віднесення такого дерева є вузли, які представляють інтервали, що фрагментують проекцію на вісь абсцис заданого регіону.

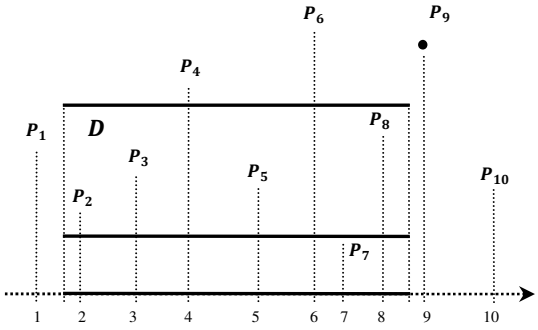


Рисунок 51. Розбиття на елементарні відрізки по осі абсцис

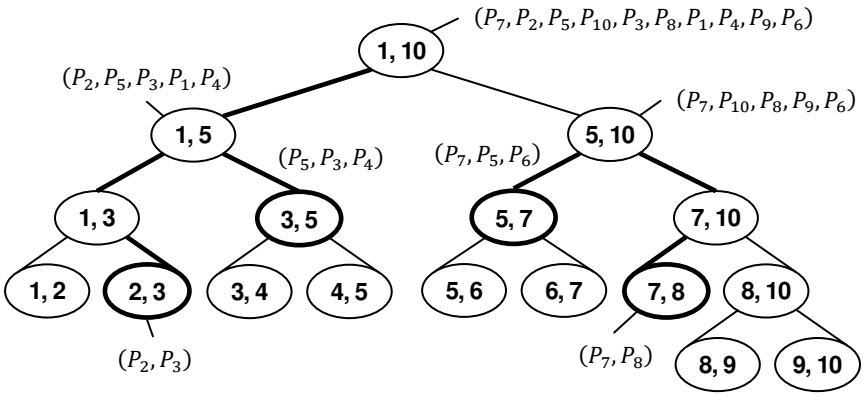


Рисунок 52. Застосування дерева регіонів

Дерево регіонів: підхід Шеймоса

Інтервали, на які розбивається початковий регіон, тепер *напіввідкриті* (вигляду $[a, b)$). Дерево регіонів (узагальнене для d вимірів) будується рекурсивно (значення всіх координат нормалізовані):

1. Початкове дерево відрізків T^* відповідає множині $\{x_1(p): p \in S\}$. Для кожного вузла v із T^* позначимо $S_d(v)$ – множину точок з S, що проектуються на відповідний відрізок по координаті x_1. Визначимо $(d-1)$-вимірну множину: $S_{d-1}(v) = \{(x_2(p), \dots, x_d(p)): p \in S_d(v)\}$.
2. Вузол v із T^* має вказівник на дерево регіонів для $S_{d-1}(v)$.

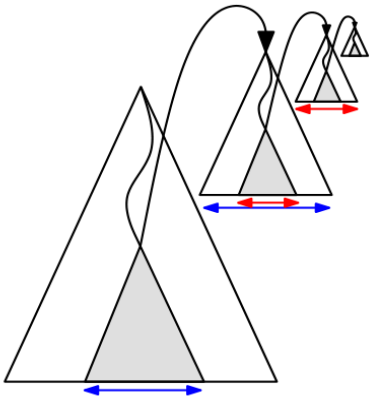


Рисунок 53. Схематичне зображення d-вимірного дерева регіонів

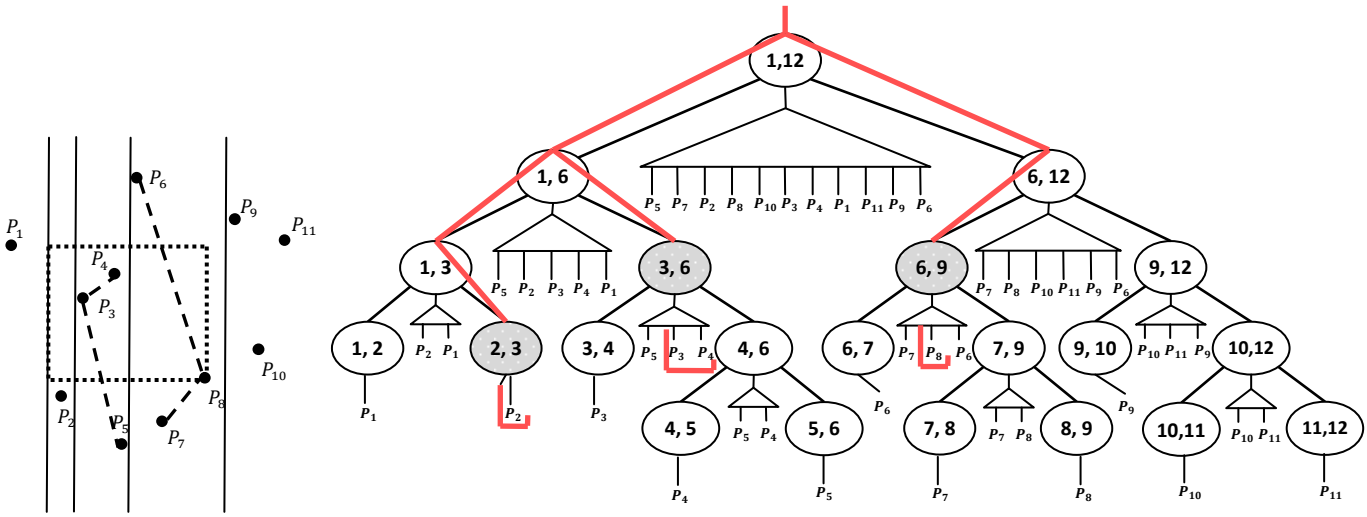


Рисунок 54. Застосування дерева регіонів. Підхід Шеймоса

Ефективність методу дерева регіонів

Теорема 16. Регіональний пошук методом дерева регіонів на d-вимірному файлі із N точок можна провести за часу запиту $O(\log^d N)$ при використанні $O(N \log^{d-1} N)$ пам'яті, якщо затратити $O(N \log^{d-1} N)$ часу на попередню обробку.

Зокрема, для $d = 2$ час запиту, пам'ять і час попередньої обробки $O(\log^2 N + k)$, $O(N \log N)$ і $O(N \log N)$ відповідно. Час пошуку можна *покращити* в логарифм разів, якщо використати техніку розшарування (*fractional cascading*). Розглянемо двовимірний випадок.

2.3.4 Техніка fractional cascading для покращення методу дерева регіонів

Кожен вузол дерева відрізків пов'язаний з двійковим деревом пошуку. Мета – уникнути другого бінарного пошуку і отримати час запиту $O(\log N + k)$. Візьмемо замість дерева *впорядкований за неспаданням список (масив)* y-координат. При цьому з'являться додаткові зв'язки між елементами масивів сусідніх рівнів.

Ідея: додати зв'язок з *першим елементом наступного масиву, який не є меншим за даний*. Якщо такого елемента немає, то вказівник нульовий. Елементи кожного наступного масиву є підмножиною елементів попереднього.

Нехай шукаємо значення в інтервалі $[20; 65]$. Спочатку бінарним пошуком в масиві A_1 знаходимо 23.

Якщо треба шукати в A_1 – рухаємось по ньому.

Якщо треба шукати в A_2 – рухаємось від знайденого ключа 23 по вказівнику *вниз* до елемента з A_2 (його ключ має значення 30) і далі йдемо по масиву (рис. 55).

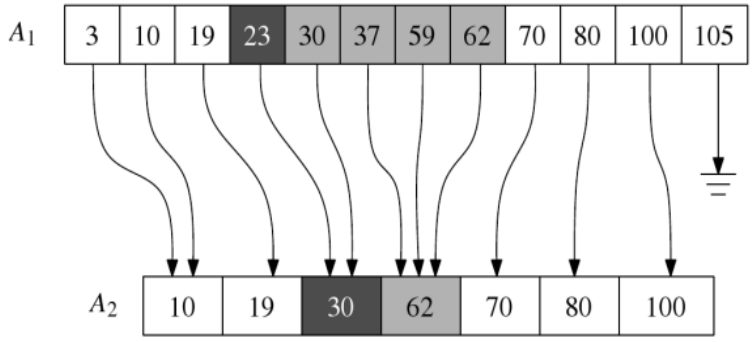


Рисунок 55. Техніка fractional cascading. Приклад для двох масивів

Важливою властивістю дерева регіонів є те, що множини ординат, асоційованих з двома потомками певного вузла, є *розбиттям* його власної множини ординат. Тому від кожного елемента масиву має відходити *два* вказівники, що вказуватимуть на перші елементи, не менші за даний, в масивах лівого та правого синів. Така модифікована структура називається *розшарованим деревом регіонів* (layered range tree) (рис. 56).

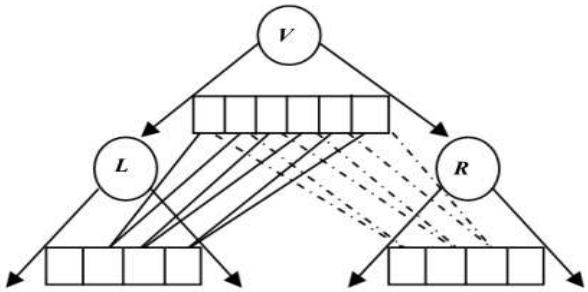


Рисунок 56. Розшароване дерево регіонів

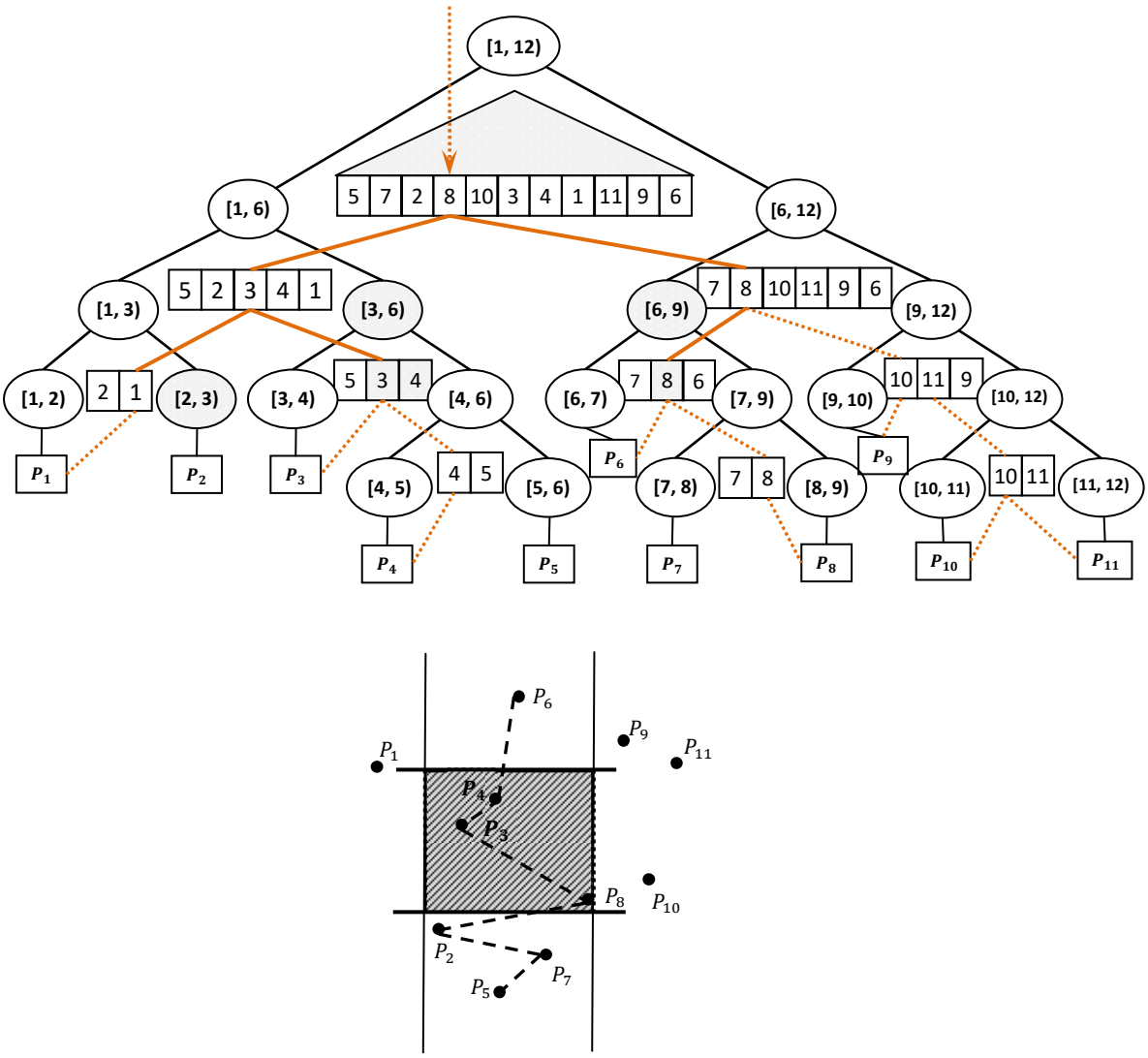


Рисунок 57. Приклад розшарованого дерева регіонів (показана лише частина зв'язків між масивами)

2.3.5 Вироджені випадки

В процесі розбиття площини можливе виникнення вироджених ситуацій, коли багато точок мають однакову x- або y-координату (рис. 58).

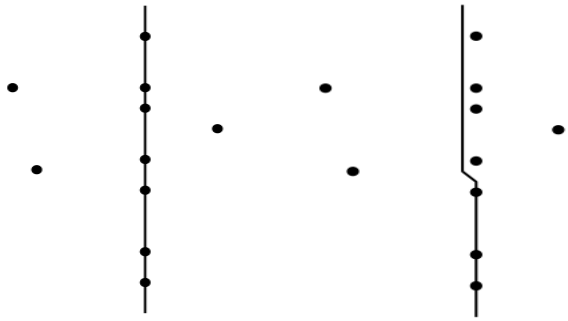


Рисунок 58. Приклад вироджених ситуацій

Перетворимо точку $p = (p_x, p_y)$ з дійсними координатами в точку, координатами якої будуть *складені числа*. Складене число $(a|b)$ – пара дійсних чисел. Введемо на них порядок: $(a|b) < (c|d) \Leftrightarrow a < c$ або ($a = c$ та $b < d$).

Нехай точка $p = (p_x, p_y)$ прийме вигляд $((p_x|p_y), (p_y|p_x))$. Тоді жодні дві точки не матимуть однакових першої чи другої координат. Регіон запиту $[x_1, x_2] \times [y_1, y_2]$ перетвориться на $[(x_1| - \infty), (x_2| + \infty) \times (y_1| - \infty), (y_2| + \infty)]$.

Умова приналежності точки регіону тепер виглядатиме так:

$$(p_x, p_y) \in [x_1, x_2] \times [y_1, y_2] \Leftrightarrow$$

$$((p_x|p_y), (p_y|p_x)) \in [(x_1| - \infty), (x_2| + \infty) \times (y_1| - \infty), (y_2| + \infty)] .$$

РОЗДІЛ 3
ЗАДАЧІ ПОБУДОВИ ОПУКЛОЇ ОБОЛОНКИ

3.1 Основна термінологія та постановка задач

Області застосування задачі надзвичайно різноманітні (рис. 58):

- розпізнавання образів;
- обробка зображень;
- статистика;
- теорія ігор;
- геоінформаційні системи;
- побудова фазових діаграм;
- статичний аналіз коду;
- як частина інших алгоритмів обчислювальної геометрії.

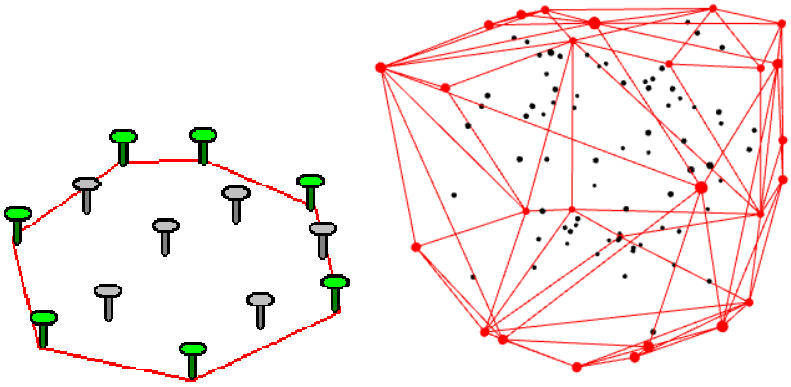


Рисунок 58. Приклади побудови опуклої оболонки для дво- та тривимірного випадку

Опукла оболонка множини точок S – найменша опукла множина, що містить S.

Нехай в просторі E^d задано k різних точок $p_1, p_2, \ldots, p_k$. Тоді множина точок p таких, що $p = a_1p_1 + a_2p_2 + \cdots + a_kp_k \ \left(a_j \in \boldsymbol{R}, a_j \geq 0, a_1 + a_2 + \cdots + a_k\right) = 1$ – *опукла множина*, породжена точками $p_1, p_2, \ldots, p_k$, а p – *опукла комбінація* $p_1, p_2, \ldots, p_k$.

Наприклад, при $k = 2$ опуклою множиною є відрізок, що з'єднує дві задані точки. Позначимо: $conv(S)$ – опукла оболонка множини S. Вважаємо S скінченною.

Для характеризації структури $conv(S)$ узагальнимо поняття опуклих многокутника й многогранника.

Поліедральна множина в E^d – це перетин скінченної множини замкнутих півпросторів (півпростір – частина E^d, розташована по один бік від деякої гіперплощини). Поліедральна множина є *опуклою*, оскільки півпростір – опукла множина, і перетин опуклих

множин теж є опуклою множиною. Приклади поліедральних множин – плоский опуклий многокутник (2D), опуклий многогранник (3D).

Скінченну d-вимірну поліедральну множину називають *опуклим d-політопом* (або просто політопом).

Теорема 16. Опукла оболонка скінченної множини точок в E^d є опуклим політопом. І навпаки, кожен опуклий політоп є опуклою оболонкою деякої скінченної множини точок.

Опуклий політоп задається описом його границі, яка складається з граней. Кожна грань опуклого політопа є опуклою множиною; k-грань означає k-вимірну грань.

Якщо політоп P має розмірність d, то його $(d-1)$-грані називаються гіпергранями, $(d-2)$-грані – підгранями, 1-грані – ребрами, а 0-грані – вершинами. Ребра і вершини залишаються собою за будь-якої розмірності. Наприклад, для 3-політопа гіперграні є плоскими многокутниками, а підграні співпадають з ребрами.

Існує співвідношення між кількістю граней і розмірністю політопа. Число $F(d,N)$ гіперграней d-політопа з N вершинами може досягати

$$F(d,N) = \begin{cases} \frac{2N}{d} \cdot \binom{N-d/2-1}{d/2-1}, & \text{для парних } d, \\ 2 \cdot \binom{N-[d/2]-1}{[d/2]}, & \text{для непарних } d. \end{cases}$$

Тобто оцінка складності $O\left(N^{[d/2]}\right)$.

При $d = 2$ політоп є опуклим многокутником і являє собою впорядковану послідовність вершин. Взагалі, будь-який многокутник (не важливо, чи опуклий) є впорядкованою послідовністю вершин. Його можна представити масивом або списком.

При $d = 3$ це – многогранник. Многогранник визначається описом своїх вершин, ребер і граней, число яких пов'язане лінійним співвідношенням (за формулою Ейлера). Тому для його представлення досить лінійної пам'яті. Більше того, скелет многогранника (множина його ребер) є планарним графом, тому можна використати одну з відповідних структур представлення такого графа.

Постановка задач опуклої оболонки

Позначимо границю опуклої оболонки через $CH(S)$.

Задача ОО1 (**ОПУКЛА ОБОЛОНКА**). В E^d задано множину S, яка містить N точок. Необхідно побудувати їх опуклу оболонку (тобто повний опис границі $CH(S)$).

Задача ОО2 (**КРАЙНІ ТОЧКИ**). В E^d задано множину S, яка містить N точок. Необхідно визначити ті з них, які є вершинами опуклої оболонки $conv(S)$.

В другій задачі немає умови впорядкованості шуканої множини точок.

Задача ОО1 асимптотично є такою ж складною, як і задача ОО2, тобто:

КРАЙНІ ТОЧКИ ∞_N ОПУКЛА ОБОЛОНКА.

Вершини многокутника, що є опуклою оболонкою, розташовані в певному порядку – можна провести паралелі з задачею сортування.

Нижня оцінка задачі пошуку опуклої оболонки: для знаходження опуклої оболонки множини з N точок у просторі $d \geq 2$ необхідно $\Omega(N \log N)$ операцій. Задачу сортування можна за N кроків звести до задачі пошуку опуклої оболонки.

Теорема 17. Задача сортування зводиться за лінійний час до задачі побудови опуклої оболонки, і для знаходження впорядкованої опуклої оболонки з N точок на площині потрібен час $\Omega(N \log N)$.

Нехай задано N додатних дійсних чисел $x_1, x_2, \dots, x_N$. Поставимо у відповідність x_i точку $(x_i\ , {x_i}^2)$ та присвоїмо їй номер i. Усі ці точки лежать на параболі $y = x^2$ (рис. 59).

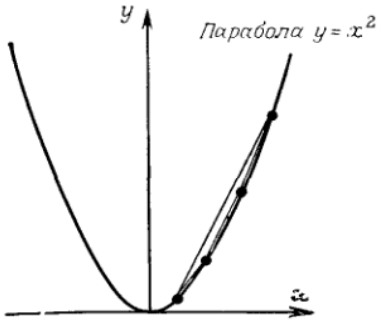


Рисунок 59. Точки $(x_i\ , {x_i}^2)$ з $x_1, x_2, \dots, x_N$ та парабола $y = x^2$

Опукла оболонка цієї множини точок, представлена у стандартному вигляді, буде складатися із списку точок множини, впорядкованого за значенням абсциси. Один перегляд цього списку дозволяє прочитати в потрібному порядку значення x_i.

Опукла оболонка буде описана послідовністю точок (вершин), x-координати яких будуть впорядковані, тому, маючи опуклу оболонку, можемо отримати впорядкований масив точок $x_1 x_2 \dots x_N$.

Нижня оцінка задачі сортування дорівнює нижній оцінці задачі пошуку опуклої оболонки і дорівнює $\Omega(N \log N)$.

Побудова опуклої оболонки – перше наближення

Зупинимось на питанні побудови опуклої оболонки на площині.

Точка p опуклої множини S називається *крайньою*, якщо не існує точок a, $b \in S$ таких, що p належить відкритому відрізку ab.

Множина E крайніх точок S є найменшою підмножиною S з властивістю $conv(E) = conv(S)$, та E точно співпадає з множиною $conv(S)$. Звідси випливає, щоб знайти опуклу оболонку, потрібно виконати наступні два кроки:

1. Визначити крайні точки (двовимірний випадок ОО2).
2. Упорядкувати ці точки так, щоб вони утворили опуклий многокутник.

Потрібна умова перевірки «крайності» точки.

Теорема 18. Точка p не є крайньою точкою плоскої опуклої множини S тоді і тільки тоді, коли вона лежить у деякому трикутнику з вершинами в точках з S, але сама не є його вершиною (рис. 60).

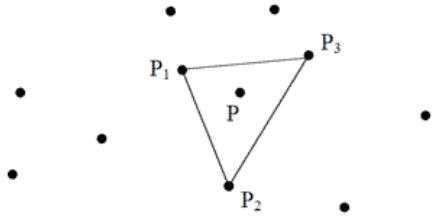


Рисунок 60. Приклад некрайної точки (тут точка P не є крайньою, бо вона знаходиться всередині трикутника $(p_1p_2p_3)$).

Всього є $O(N^3)$ трикутників, що визначаються N точками множини S, тому за час $O(N^3)$ можна сказати, чи є дана точка крайньою. Таким чином, процедура перевірки на «крайність» для всіх N точок множини S потребує часу $O(N^4)$.

Тепер потрібно «упорядкувати» знайдені крайні точки в опуклу оболонку.

Теорема 19. Промінь, що виходить із внутрішньої точки опуклої обмеженої множини S, перетинає її границю рівно в одній точці.

Теорема 20. Послідовні вершини опуклого многокутника впорядковані за полярним кутом відносно довільної внутрішньої точки (рис. 61).

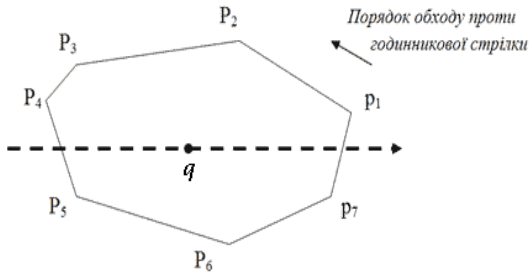


Рисунок 61. Вершини многокутника P впорядковані відносно точки q.

За відомими крайніми точками множини опуклу оболонку можна знайти, вибравши точку q як внутрішню точку оболонки та впорядкувавши потім крайні точки відповідно до полярного кута відносно q.

Способи знаходження точки q:

1. За точку q можна взяти центроїд множини крайніх точок $p_1, p_2, \dots, p_k$: $p = (x_p, y_p)$, де $x_p = (\sum x_i)/k$, $y_p = (\sum y_i)/k$ (відомо, що центроїд множини точок є внутрішньою точкою опуклої оболонки). Центроїд множини із N точок в k-вимірному просторі може бути тривіально визначений за $O(Nk)$ арифметичних операцій.
2. (Грехем). Достатньо взяти центроїд трьох довільних неколінеарних точок, починаючи з двох довільних точок, і по черзі досліджувати $N-2$ точки, що лишилися, шукаючи серед них одну, яка не лежить на прямій, що визначається першими двома точками.

В гіршому випадку для цього потрібен час $O(N)$, але в реальності все скоріше закінчиться за фіксований час.

Отже, знайшовши крайні точки множини S, можна за час $O(N)$ знайти точку q, яка є внутрішньою точкою оболонки.

Залишається відсортувати (впорядкувати) крайні точки відповідно до значення полярного кута, використовуючи точку q як початок системи координат. Це можна зробити, перетворивши точки до полярної системи координат, за час $O(N)$, а потім відсортувавши їх за час $O(N \log N)$. Насправді явне обчислення полярних координат не потрібне, оскільки при сортуванні достатньо знати, який з двох кутів, що порівнюються, більший (а не конкретні їх величини).

Нехай на площині задані дві точки p_1 і p_2. Точка p_2 утворює з віссю $0X$ строго *менший* полярний кут, ніж p_1, тоді і лише тоді, коли трикутник $(0, p_2, p_1)$ має строго додатну площу (рис. 62).

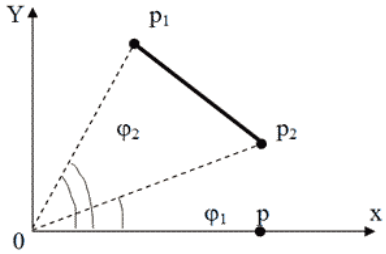


Рисунок 62.

Таким чином, отримали перший примітивний алгоритм побудови опуклої оболонки множини точок, що працюватиме за час $O(N^4)$ і буде використовувати тільки арифметичні операції та порівняння.

3.2 Метод Грехема

Нехай внутрішня точка вже знайдена, і вона є початком координат. Впорядкуємо лексикографічно N точок відповідно до значень полярного кута і відстані від початку координат. При виконанні сортування реальні відстані між точками ніде не обчислюються, бо порівнюються кути.

Єдиний випадок, коли треба порівняння відстаней: якщо дві точки мають однаковий полярний кут. Але тоді вони лежать на одній прямій з початком координат, і порівняння в цьому випадку тривіальне.

Впорядковані вершини організовані у двозв'язний циклічний список. Якщо точка не є вершиною опуклої оболонки, то вона є внутрішньою точкою для деякого трикутника $(0pq)$, де p і q – послідовні вершини опуклої оболонки.

Суть алгоритму Грехема полягає в однократному перегляді впорядкованої послідовності точок, в процесі якого вилучаються внутрішні точки. Точки, що лишилися, є вершинами оболонки, які подано у відповідному порядку.

Перегляд починається з точки, яку позначено як ПОЧАТОК. У якості цієї точки можна взяти найправішу з найменшою ординатою точку з даної множини, яка точно буде вершиною опуклої оболонки (рис. 63)

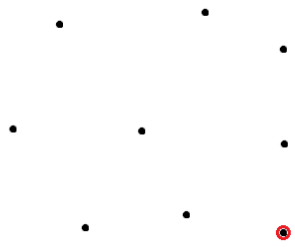


Рисунок 63. Найправіша з найменшою ординатою точка множини

Трійки послідовних точок багатократно перевіряються у порядку обходу проти годинникової стрілки з метою визначення, чи утворюють вони кут, більший або рівний π.

Якщо внутрішній кут $p_1p_2p_3 \geq \pi$, то $p_1p_2p_3$ утворюють «правий поворот» $(\Delta \geq 0)$, інакше $p_1p_2p_3$ утворюють «лівий поворот» $(\Delta < 0)$, де

$$\Delta = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_2 & 1 \end{vmatrix}.$$

При такому обході опуклого многокутника будуть робитися лише «ліві повороти».

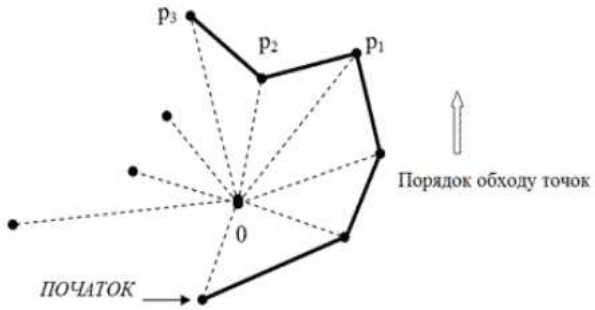


Рисунок 64. Обхід опуклого многокутника

Якщо $p_1p_2p_3$ утворюють правий поворот, то p_2 не може бути крайньою точкою, оскільки вона буде внутрішньою для трикутника $(0p_1p_3)$. Тому вершина p_2 має бути вилучена.

Таким чином, залежно від результатів перевірки кута, який утворюється трійкою вершин, можливі два варіанти продовження перегляду:

1) $p_1p_2p_3$ утворюють правий поворот. Вилучити вершину p_2 і перевірити трійку $p_0p_1p_3$.
2) $p_1p_2p_3$ утворюють лівий поворот. Продовжити перегляд, перейшовши до трійки $p_2p_3p_4$.

Перегляд завершується тоді, коли, обійшовши всі вершини, знову приходимо у вершину ПОЧАТОК. Слід зауважити, що вершина ПОЧАТОК ніколи не видаляється, бо вона є крайньою точкою.

Розглянутий метод обходу границі многокутника назвемо *обходом Грехема (Graham scan).*

Алгоритм обходу Грехема. Після завершення алгоритму список містить впорядковані вершини оболонки.

procedure ОБОЛОНКА_ГРЕХЕМА(S)

1. Знайти внутрішню точку q.
2. Використовуючи q як початок координат, упорядкувати точки множини S лексикографічно відповідно до полярного кута і відстані від q. Організувати точки множини у вигляді циклічного двозв'язного списку з посиланнями НАСТУП і ПОПЕР і вказівником ПОЧАТОК на першу вершину. Значення $true$ логічної змінної f вказує на те, що вершина ПОЧАТОК виявилася досягнутою при прямому просуванні по оболонці, а не в результаті повернення.
3. (Обхід)

```
begin
  v := ПОЧАТОК;
  w := ПОПЕР[v];
```

```
    f := false;
    while (НАСТУП[v] ≠ ПОЧАТОК or f = false) do
      begin
        if (НАСТУП[v] = w)
          then f := true;
        if (три точки v, НАСТУП[v], НАСТУП[НАСТУП[v]]
          утворюють "лівий поворот")
            then v := НАСТУП[v]
            else
              begin
                ВИЛУЧИТИ НАСТУП[v];
                v := ПОПЕР[v];
              end;
      end;
end.
```

Теорема 21. Опукла оболонка N точок на площині може бути знайдена методом Грехема за час $O(N \log N)$ при використанні пам'яті $O(N)$ з використанням лише арифметичних операцій і порівнянь.

Доведення. Із попереднього обговорення алгоритму зрозуміло, що в ньому використовуються лише арифметичні операції та порівняння. Кроки 1 і 3 потребують лінійного часу, тоді як крок 2 (сортування), який є визначальним за часом роботи, виконується за час $O(N \log N)$. Для представлення зв'язного списку точок достатньо $O(N)$ пам'яті.

3.2.1 Метод Ендрю

Розглянемо модифікацію підходу, що дозволяє уникнути підрахунків полярних координат. Визначимо на заданій множині з N точок її ліву та праву крайні точки l та r. Побудуємо пряму через ці дві точки.

Решта точок розбиваються на дві підмножини: нижню (нижче прямої) і верхню (вище прямої). Нижня підмножина породжує ламану (*нижню оболонку*), монотонну відносно осі $0X$. Аналогічно верхня підмножина породжує *верхню оболонку*.

Об'єднання цих ламаних дасть опуклу оболонку початкової множини.

Розглянемо побудову верхньої оболонки. Точки впорядковуються за зростанням абсциси. До отриманої послідовності застосовують обхід Грехема (рис. 65).

Фактично отримали частковий випадок для методу Грехема, коли точка q вибирається нескінченно віддаленою по осі $0Y$ в напрямку $-\infty$, так що впорядкованість за абсцисою співпадає з впорядкованістю за полярним кутом.

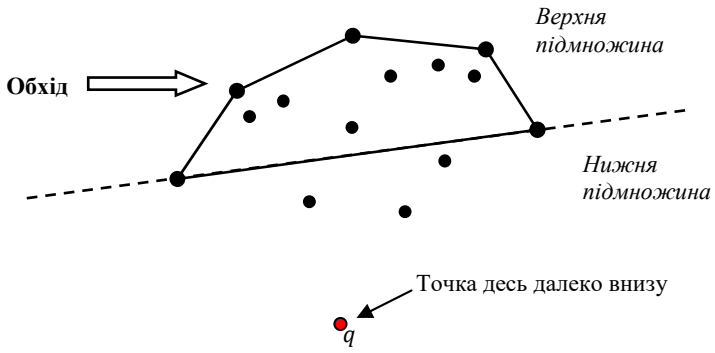


Рисунок 65. Застосування обходу Грехема у методі Ендрю

Особливості алгоритму Грехема:

- оптимальний в найгіршому випадку;
- безпосередньо пов'язаний із сортуванням;
- застосовується тільки на площині, не має узагальнення на більші розмірності;
- не є відкритим алгоритмом (перед початком роботи повинні бути відомі всі точки);
- не дозволяє розпаралелити вихідну задачу.

Розглянемо приклад. Для заданої множини точок $p_1, p_2, p_3, p_4, p_5, p_6, p_7$ застосувати обхід Грехема.

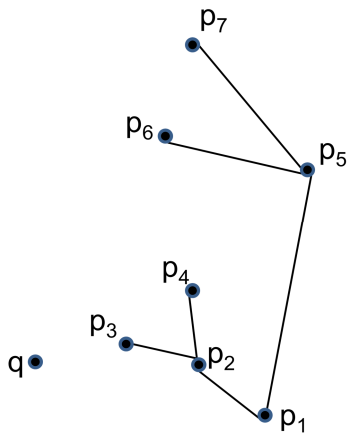


Рисунок 66. Приклад застосування обходу Грехема для точок $p_1, p_2, p_3, p_4, p_5, p_6, p_7$

- $p_1p_2p_3\ (+)$
- $p_2p_3p_4\ (-)$
- $p_1p_2p_4\ (-)$
- $p_1p_4p_5\ (-)$
- $p_1p_5p_6\ (+)$
- $p_5p_6p_7\ (-)$
- $p_1p_5p_7\ (+)$
- $p_5p_7p_1\ (+)$

3.3 Метод Джарвіса

Многокутник можна задавати не тільки через вершини, а й через ребра. Альтернативний підхід до побудови опуклої оболонки: замість шукати крайні точки, спробувати визначити ребра оболонки.

Теорема 22. Відрізок l, який визначається двома точками, є ребром опуклої оболонки ⇔ коли усі інші точки заданої множини лежать на цьому відрізку або по один бік від нього.

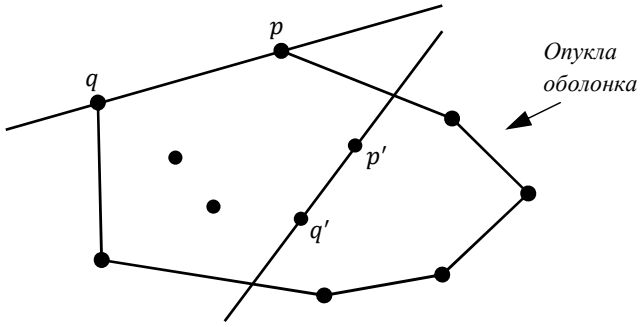


Рисунок 67. Приклад ребра та не ребра опуклої оболонки (pq є ребром, $p'q'$ – не є ребром)

Теорема 22. Якщо знайдено відрізок $l = pq$, $p, q \in S$, $pq \in CH(S)$ ⇔ для будь-якого $r \in S$, pq – є ребром опуклої оболонки, $S_{p,r,q} \leq 0$, де $S_{p,r,q}$ – орієнтована площа трикутника.

Припустимо, що знайдено найменшу в лексикографічному порядку точку p_1 заданої множини точок. Ця точка явно є вершиною оболонки. Треба знайти наступну за нею вершину p_2 опуклої оболонки. Точка p_2 – це точка, яка має найменший додатний полярний кут відносно точки p_1 як початку координат.

Аналогічно наступна точка p_3 має найменший полярний кут відносно точки p_2 як початку координат, і кожна наступна точка опуклої оболонки може бути знайдена за лінійний час.

Алгоритм Джарвіса обходить кругом опуклу оболонку, породжуючи в необхідному порядку послідовність крайніх точок, по одній точці на кожному кроці. Так будується частина опуклої оболонки (ламана лінія) від найменшої в лексикографічному порядку точки (p_1) до найбільшої в лексикографічному порядку точки (p_5).

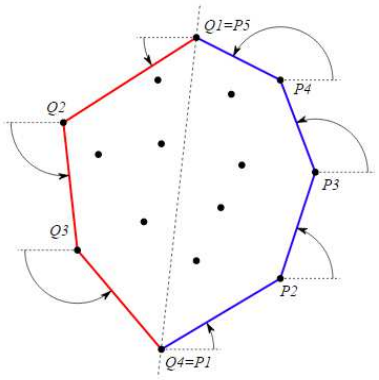


Рисунок 68. Обхід опуклої оболонки за алгоритмом Джарвіса (обхід Джарвіса)

Побудова опуклої оболонки завершується знаходженням іншої ламаної, яка виходить із найбільшої в лексикографічному порядку точки до найменшої.

Виходячи із симетричності цих двох етапів, необхідно змінити на протилежні напрямки осей координат і мати справу з полярними кутами, найменшими відносно від'ємного напрямку осі x.

Круговий обхід опуклої оболонки, здійснений даним алгоритмом, називають *обходом Джарвіса (Jarvis march)*.

Алгоритм Джарвіса витрачає на знаходження кожної точки оболонки лінійний час.

Оскільки усі N точок множини можуть лежати на її опуклій оболонці (бути її вершинами), то час виконання алгоритму в найгіршому випадку складе $O(N^2)$, що гірше, ніж в алгоритмі Грехема.

Якщо в дійсності число вершин опуклої оболонки дорівнює h, то час виконання алгоритму Джарвіса буде $O(hN)$, і він дуже ефективний, коли апріорі відомо, що значення h мале. Наприклад, якщо оболонка є многокутником з довільним постійним числом сторін, то її можна знайти за лінійний відносно числа точок час.

Ідея пошуку послідовних вершин оболонки через багатократне визначення мінімального кута викликає асоціацію із «загортанням» двовимірного предмета або обтягуванням шнурком групи вбитих у дошку гвіздків (рис. 69).

Метод Джарвіса можна розглядати як двовимірний варіант підходу, заснованого на ідеї «загортання подарунку».

Такий підхід може застосовуватись і в просторах з розмірністю більше двох.

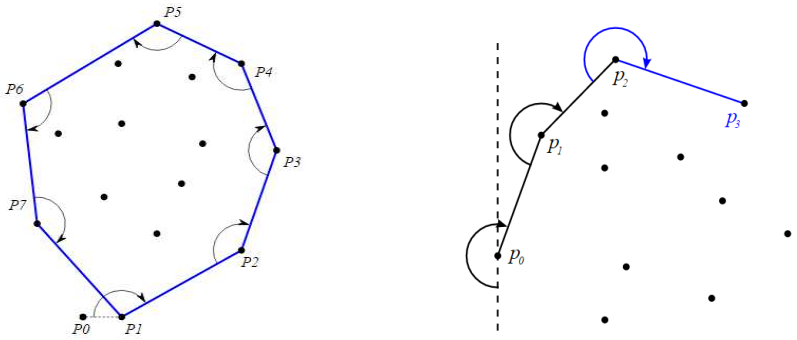


Рисунок 69. Варіації реалізації обходу Джарвіса

3.4 «Швидкий» метод побудови опуклої оболонки

Один з методів, що використовує рекурсію. В основі лежить підхід, схожий на той, що використовується в швидкому сортуванні. Тому і назва схожа: QuickHull («ШвидкОбол»).

Метод розбиває множину S із N точок на дві підмножини, кожна з яких міститиме одну із двох ламаних, з'єднання яких дає многокутник опуклої оболонки.

Початкове розбиття множини визначається прямою, яка проходить через дві точки l і r, з найменшою і найбільшою абсцисами відповідно.

Позначимо: $S^{(1)}$ – підмножина точок, що розташовані вище або на прямій, яка проходить через l і r; $S^{(2)}$ – симетричним чином визначена підмножина точок, розташованих нижче або на тій самій прямій (рис. 70).

Зауваження. Строго кажучи, $\{S^{(1)}, S^{(2)}\}$ не є розбиттям множини S.

На всіх наступних етапах відбувається типова обробка підмножин.

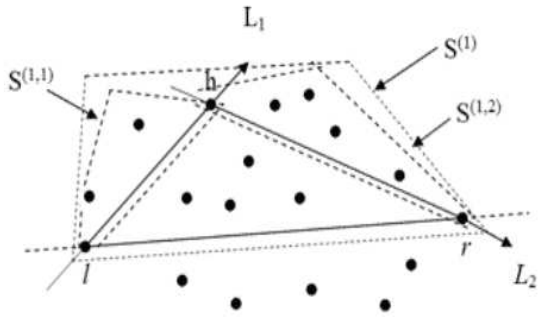


Рисунок 70. Розбиття множини S

Розглянемо верхню підмножину $S^{(1)}$.

Визначимо точку h, для якої трикутник (hlr) має максимальну площу серед усіх трикутників $\{(plr): p \in S^{(1)}\}$. Якщо таких точок більше однієї, то вибираємо ту, в якої кут $\angle(hlr)$ більший (тобто найлівішу).

Точка h гарантовано належить опуклій оболонці. Справді, проведемо через точку h пряму, паралельну відрізку lr. Вище цієї прямої не буде точок з множини $S^{(1)}$.

Можливо, на прямій знайдуться інші точки з $S^{(1)}$, відмінні від h. Але завдяки нашому вибору точка h буде з них най лівішою (рис. 71).

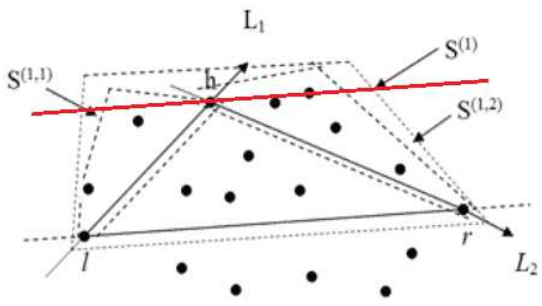


Рисунок 71. Обрана точка h буде найлівішою

Побудуємо дві прямі: L_1 – спрямована із l в h; L_2 – спрямована із h в r.

Для кожної точки множини $S^{(1)}$ визначається її положення відносно цих прямих. Важливо відмітити, що жодна з точок не знаходиться одночасно ліворуч від L_1 та від L_2.

Всі точки, розташовані праворуч від обох прямих, є внутрішніми точками трикутника (lrh) і тому можуть бути вилучені із подальшої обробки.

Точки, розташовані ліворуч від L_1 або на ній (і розташовані праворуч від L_2), утворюють підмножину $S^{(1,1)}$. Аналогічно утворюється підмножина $S^{(1,2)}$.

Утворені підмножини $S^{(1,1)}$ і $S^{(1,2)}$ передаються на наступний рівень рекурсивної обробки. Опукла оболонка для $S^{(1)}$ утворюється склейкою впорядкованих списків вершин опуклих оболонок для $S^{(1,1)}$ і $S^{(1,2)}$.

За аналогічним принципом обробляється нижня підмножина $S^{(2)}$.

Розглянутий метод використовує як примітивні функції обчислення площі трикутника і визначення положення точки відносно прямої.

Виберемо початкові значення $\{l_0, r_0\}$ точок l та r:

- за l_0 – точку з найменшою абсцисою (x_0, y_0);
- за r_0 – точку $(x_0, y_0 - \varepsilon)$, де $\varepsilon > 0$ – мале число.

За початкову пряму, яка розбиває множину на частини, обирається вертикальна пряма, що проходить через точку l_0.

Після завершення алгоритму точка r_0 вилучається (покладається $\varepsilon = 0$).

Припустимо, S містить хоча б дві точки, а функція НАЙДАЛЬШАТОЧКА$(S; l, r)$ знаходить точку h описаним вище способом.

Результат ШВИДКОБОЛу – впорядкований список точок; «*» означає конкатенацію списків.

function ШВИДКОБОЛ$(S; l, r)$
  begin
    if $(S = \{l, r\})$ **then**
      return (l, r) (**опукла оболонка складається із єдиного орієнтованого ребра**)
    else begin
      $h :=$ НАЙДАЛЬШАТОЧКА$(S; l, r)$;
      $S^{(1)} :=$ точки множини S , розташовані ліворуч від $[l, h]$ або на ній;
      $S^{(2)} :=$ точки множини S , розташовані ліворуч від $[h, r]$ або на ній;
      return ШВИДКОБОЛ$(S^{(1)}; l, r) \cdot$ (ШВИДКОБОЛ$(S^{(2)}; h, r) - h)$
    end;
end.

Таким чином, початковий виклик ШВИДКОБОЛу матиме вигляд:

begin
  $l_0 = (x_0, y_0) :=$ точка множини S з найменшою абсцисою;

$r_0 = (x_0, y_0 - \varepsilon)$;
ШВИДКОБОЛ$(S; l_0, r_0)$;
вилучити точку r_0 (**еквівалентно тому, що покласти $\varepsilon = 0$**)
end.

Оцінка складності. При заданих N точках для виділення з множини S підмножин $S^{(1)}$ та $S^{(2)}$ (з неявним видаленням точок, що потрапили всередину трикутника (lrh)) треба $O(N)$ операцій. Потім рекурсивно викликаються обробки $S^{(1)}$ та $S^{(2)}$.

Потужність кожної з цих підмножин не перевищує потужності S, помноженої на деяку константу < 1; така умова виконується на кожному рівні рекурсії.

Тому загальний час роботи алгоритму $O(N \log N)$. Однак в найгіршому випадку час складе $O(N^2)$. Складність алгоритму така сама, як в QuickSort.

Переваги методу:

1) піддається розпаралеленню;
2) можливе узагальнення на більші розмірності.

Недоліки методу:

1) квадратична складність в найгіршому випадку (як і в швидкому сортуванні);
2) підзадачі, на які розбивається задача, можуть мати зовсім різні розміри (тобто не реалізується принцип збалансованого розбиття при підході «розділяй та владарюй»).

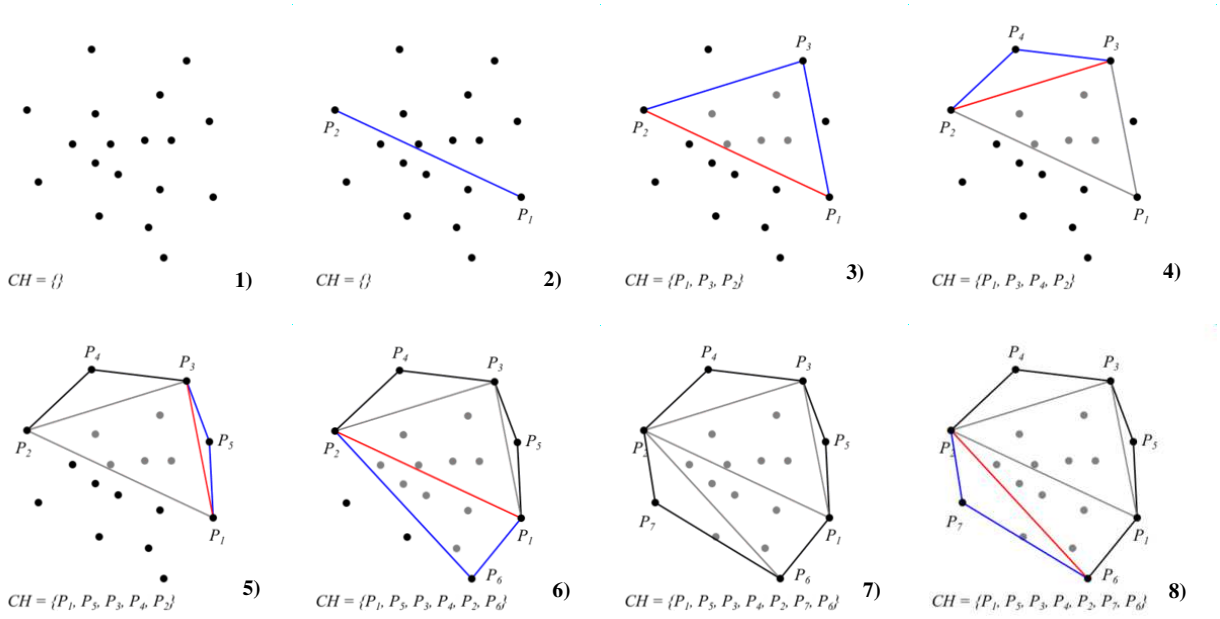


Рисунок 72. Приклад виконання «швидкого» методу побудови опуклої оболонки

3.5 Алгоритм типу «розділяй та владарюй»

Нехай при розв'язанні задачі побудови опуклої оболонки початкова множина точок була розбита на дві частини (S_1 і S_2), кожна з яких містить половину точок початкової множини.

Якщо тепер рекурсивно знайдені окремо $CH(S_1)$ та $CH(S_1)$, то які додаткові витрати необхідні для побудови $CH(S_1 \cup S_2)$, тобто опуклої оболонки початкової множини?

Для відповіді на це питання можна скористатись таким співвідношенням:

$$CH(S_1 \cup S_2) = CH\big(CH(S_1) \cup CH(S_2)\big).$$

Ідея в тому, що $CH(S_1)$ та $CH(S_2)$ – це не просто невпорядковані множини точок, а опуклі многокутники (рис. 73).

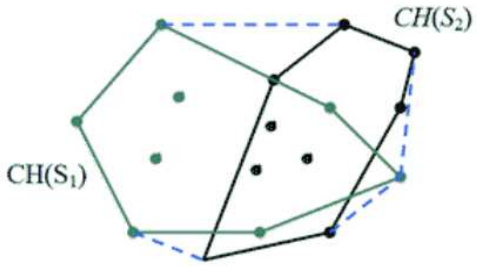


Рисунок 73. Опуклі многокутники $CH(S_1)$ та $CH(S_2)$

Задача ООЗ (ОПУКЛА ОБОЛОНКА ОБ'ЄДНАННЯ ОПУКЛИХ МНОГОКУТНИКІВ). Задано два опуклих многокутники P_1 і P_2. Знайти опуклу оболонку їх об'єднання.

Загальна схема розв'язання цієї задачі злиття.

procedure ЗЛИОБОЛ(S)

1. Якщо $|S| \leq k_0$ (k_0 – деяке невелике ціле число), то побудувати опуклу оболонку одним із прямих методів і зупинитись; інакше перейти до кроку 2.
2. Розбити початкову множину S довільним чином на дві приблизно рівні за потужністю підмножини S_1 і S_2.
3. Рекурсивно знайти $CH(S_1)$ і $CH(S_2)$.
4. Злити (об'єднати) дві отримані оболонки разом, утворюючи $CH(S)$.

Оцінимо час пошуку опуклої оболонки для множини з N точок $T(N)$.

Позначимо через $U(N)$ час, необхідний для знаходження опуклої оболонки об'єднання опуклих многокутників, кожен з яких має $N/2$ вершин. Враховуючи, що $CH(S_1 \cup S_2) = CH\big(CH(S_1) \cup CH(S_2)\big)$ маємо: $T(N) \leq 2T(N/2) + U(N)$.

Для розв’язку цього співвідношення уточнимо алгоритм злиття.

Алгоритм злиття Шеймоса

procedure ЗЛИОБОЛ_ОПУКЛ_МНОГОКУТН(P_1, P_2)

Крок 1. Знайти деяку внутрішню точку p многокутника P_1.

(Наприклад, центроїд трьох довільних вершин P_1. Така точка p буде внутрішньою точкою $CH(P_1 \cup P_2)$.

Крок 2. Визначити, чи є p внутрішньою точкою P_2. Це може бути зроблено за час $O(N)$. Якщо p не є внутрішньою точкою P_2, перейти до кроку 4.

Крок 3. Нехай p є внутрішньою точкою P_2.

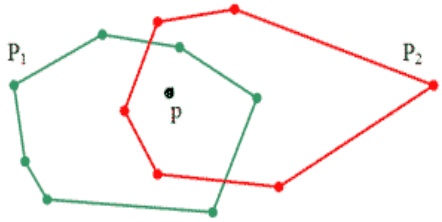


Рисунок 74. Точка p є внутрішньою точкою P_2

За теоремою 20 вершини як P_1, так і P_2 виявляються впорядкованими відповідно до значення полярного кута відносно внутрішньої точки p.

За час $O(N)$ можна отримати впорядкований список вершин як P_1, так і P_2 шляхом злиття списків вершин цих многокутників.

Перейти до кроку 5.

Крок 4. Нехай p не є внутрішньою точкою P_2.

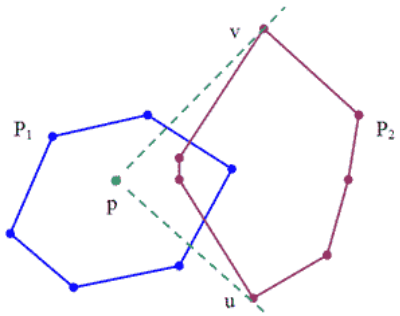


Рисунок 75. Точка p не є внутрішньою точкою P_2

Якщо дивитись із точки p, то многокутник P_2 лежить у клині з кутом розвороту, який є меншим або рівним π.

Цей клин визначається двома вершинами u та v многокутника P_2, які можуть бути знайдені за лінійний час за один обхід вершин многокутника P_2.

Ці вершини розбивають границю P_2 на два ланцюги вершин, які є монотонними відносно зміни полярного кута з початком в p.

Один із цих двох ланцюгів (опуклий за напрямком до точки p) може бути відразу вилучений, оскільки його вершини будуть внутрішніми точками $CH(S_1 \cup S_2)$. Інший ланцюг P_2 і границя P_1 є двома впорядкованими списками, що містять сумарно не більше N вершин.

а час $O(N)$ їх можна злити в один список вершин $P_1 \cup P_2 2$, впорядкованих за кутом відносно точки p.

Крок 5. До отриманого списку можна застосувати метод обходу Грехема, який вимагає лише лінійного часу. Результат – опукла оболонка $P_1 \cup P_2$.

Оцінка алгоритму. Якщо многокутник P_1 має m вершин, а P_2 має n вершин, то час виконання цього алгоритму дорівнює $O(m+n)$, що є оптимальним. Отже, час злиття лінійний, і $U(N) = O(N)$. В цьому випадку розв'язок розглянутого вище рекурентного співвідношення $T(N) = O(N \log N)$.

Теорема 23. Опукла оболонка об'єднання двох опуклих многокутників може бути знайдена за час, пропорційний сумарному числу їх вершин.

Опорною прямою до опуклого многокутника P називається пряма l, яка проходить через деяку вершину многокутника P таким чином, що внутрішня частина P повністю знаходиться по одну сторону від l. В певному розумінні поняття опорної прямої аналогічне поняттю дотичної.

Очевидно, що два опуклих многокутники P_1 і P_2 з n та m вершинами відповідно, таких, що один не лежить всередині іншого, мають загальні опорні прямі (принаймні не більше $2\min(n,m)$).

Якщо вже побудована опукла оболонка об'єднання P_1 і P_2, то опорні прямі обчислюються в результаті перегляду списку вершин $CH(P_1 \cup P_2)$. Кожна пара послідовних вершин $CH(P_1 \cup P_2)$, одна з яких належить P_1, а інша P_2, визначає опорну пряму.

3.6 Динамічні алгоритми побудови опуклої оболонки

У кожному із розглянутих алгоритмів вимагалось, щоб усі дані, які обробляються (тобто точки), були відомі до початку роботи алгоритму. В багатьох прикладних областях, де виникають геометричні задачі, зокрема пов'язані з обробкою даних в реальному часі, ряд обчислень повинен виконуватись в міру надходження точок.

Алгоритм, що обробляє дані в міру їх надходження, називається *відкритим*, а алгоритм, який обробляє всю сукупність в цілому, називається *закритим*. Іноді вважають, що кожен наступний елемент надходить після завершення обробки попереднього, але таке може бути не завжди.

Назвемо часовий інтервал між вводом двох послідовних елементів даних *затримкою надходження даних*. Вважатимемо, що надходження даних відбувається рівномірно в часі. В такому випадку корекція має виконуватися за час, що не перевищує постійної затримки надходження даних. Алгоритми, що працюють у такому режимі, називаються відповідно алгоритмами реального часу.

Загалом відкриті алгоритми будуть глобально менш ефективними за закриті (плата за відкритість).

Задача ОО4 (ВІДКРИТИЙ АЛГОРИТМ ДЛЯ ОПУКЛОЇ ОБОЛОНКИ). На площині задана послідовність із N точок $p_1, \dots, p_N$. Треба знайти їх опуклу оболонку, організувавши обробку таким чином, щоб після обробки точки p_i мали $CH(\{p_1, \dots, p_i\})$.

Задача ОО5 (ОПУКЛА ОБОЛОНКА В РЕАЛЬНОМУ ЧАСІ). На площині задана послідовність із N точок $p_1, \dots, p_N$. Треба знайти їх опуклу оболонку за умови, що час затримки надходження точок постійний.

Спробуємо розробити відкритий алгоритм побудови опуклої оболонки, взявши за основу алгоритм Грехема:

1. Вводити точки до тих пір, поки не будуть знайдені три неколінеарні точки. Їх центроїд буде внутрішньою точкою кінцевої опуклої оболонки і, отже, підходить для початку координат, відносно якого визначаються полярні кути точок при сортуванні в алгоритмі ОБОЛОНКА ГРЕХЕМА.
2. Підтримувати зв'язаний список упорядкованих крайніх точок. При надходженні точки p_i тимчасово вставити її в цей список відповідно до її полярного кута, витративши на це час $O(i)$.
3. Виконати перегляд зв'язаного списку крайніх точок методом Грехема. Оскільки обхід методом Грехема лінійний, то необхідно лише $O(i)$ часу. Можливі три варіанти завершення цього кроку:
   - точка p_i є крайньою для обробленої на даний момент множини, і її включення до числа крайніх точок спричиняє вилучення декількох інших точок;
   - точка p_i є крайньою, але ніякі інші точки не вилучаються;
   - точка p_i є внутрішньою для опуклої оболонки, що створюється, і тому вона вилучається.

Загальний час виконання цього алгоритму в гіршому випадку $O(N^2)$ (якщо кожна точка є крайньою).

Шеймос покращив наведений алгоритм, щоб він виконувався для множини із N точок глобально за оптимальний час $O(N \log N)$.

Суть ідеї: на кроці 2 замість лінійної вставки робиться бінарна і відповідно до цього на кроці 3 більш ефективно (логарифмічно) виконується пошук замість використання лінійного перегляду методом Грехема. Але при цьому на корегування треба витратити час $O(\log N^2)$. Однак чи буде цей оптимізований алгоритм достатньо хороший для обробки в реальному часі?

Слід зауважити, що нижні оцінки, одержані для закритих алгоритмів, в рівній мірі можуть бути застосовані і до відкритих алгоритмів.

Теорема 24. Будь-який відкритий алгоритм побудови опуклої оболонки в найгіршому випадку витрачає на обробку між надходженнями послідовних елементів даних час $\Omega(\log N)$.

Таким чином, навіть після модифікації Шеймоса згаданий алгоритм не задовольняє вимогам, які висуваються до алгоритмів реального часу.

Препарата розробив алгоритм з оптимальною глобальною часовою оцінкою виконання і часом корекції $O(\log N)$.

3.6.1 Алгоритм Препарати

В основі алгоритму лежить вміння ефективно будувати дві опорні прямі до опуклого многокутника, що проходять через деяку точку.

Нехай точки обробляються в порядку $p_1, p_2, \dots, p_{i-1}$ та p_i – поточна точка. Позначимо через C_{i-1} опуклу оболонку множини $\{p_1, p_2, \dots, p_{i-1}\}$. Необхідно побудувати з точки p_i опорні прямі до C_{i-1} (якщо вони існують).

Опорних прямих не існує $\Leftrightarrow p_i$ – внутрішня точка. Тоді вона просто видаляється. Інакше повинна вилучитись відповідна низка вершин, яка міститься між двома опорними точками, а замість них вставляється точка p_i (рис. 76).

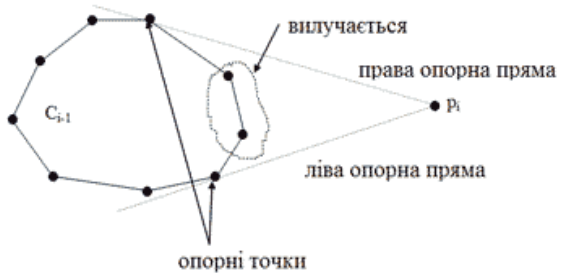


Рисунок 76. Опорні прямі з точки p_i до опуклого многокутника C_{i-1}

Для зручності опорну пряму будемо називати *лівою* або *правою* відповідно до того, по яку сторону вона знаходиться, якщо дивитися з точки p_i на C_{i-1}.

Класифікація вершин многокутника C відносно відрізка $[p, v]$, який з'єднує вершину v з точкою p:

- *Ввігнута*: відрізок $[p, v]$ перетинає внутрішню частину многокутника C (рис. 77).

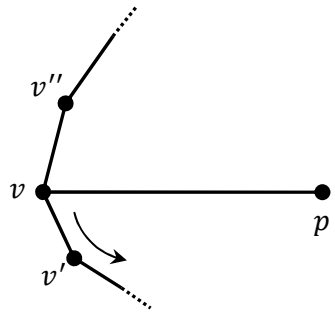


Рисунок 77. Ввігнута вершина (стрілкою позначено напрям обходу при пошуку лівої опорної прямої)

- *Опорна*: дві суміжні з v вершини лежать по один бік від прямої, яка проходить через точки p і v (рис. 78).

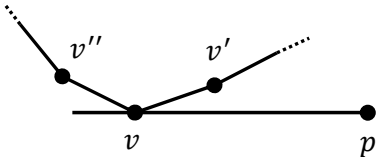


Рисунок 78. Опорна вершина

- *Опукла*: інакше (рис. 79).

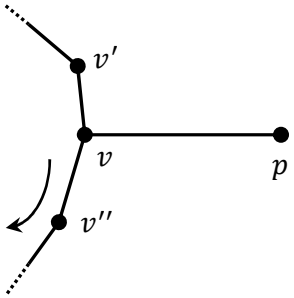


Рисунок 79. Опукла вершина (стрілкою позначено напрям обходу при пошуку лівої опорної прямої)

Класифікація однієї вершини займе константний час. Якщо v – опорна вершина, то на цьому розв'язання задачі завершується. Нехай шукається ліва опорна пряма. Якщо точка v не є опорною, то необхідно рухатися по вершинам многокутника C проти або за годинниковою стрілкою залежно від того, чи є v ввігнутою або опуклою вершиною.

Таким способом можна визначити дві опорні точки (якщо вони існують). Потім треба мати можливість вилучити із циклічної послідовності вершин многокутника C низку вершин (можливо, порожню) і вставити в утворений розрив точку p.

При цьому виконуються такі операції:

I. ПОШУК в упорядкованій послідовності елементів (циклічного списку вершин оболонки) для визначення опорних прямих з точки p_i;

II. РОЗЧЕПЛЕННЯ послідовності на дві послідовності і ЗЧЕПЛЕННЯ двох послідовностей;

III. ВСТАВКА одного елемента (поточної точки p_i).

Структура даних, яка цілком задовольняє перераховані вимоги – *зчеплена черга*.

Вона реалізується за допомогою збалансованого за висотою дерева пошуку, і при цьому кожна із вказаних операцій виконується за час $O(\log i)$ в найгіршому випадку, де i – число вузлів дерева.

Кільцева послідовність вершин зображується в цій деревовидній структурі даних ланцюгом, і при цьому перший і останній елементи вважаються суміжними.

В структурі T виділяються дві вершини : m – найлівіший член ланцюга і M – кореневий член ланцюга. Також використовується кут $\angle(mp_iM) = \alpha$. Цей кут називається *опуклим*, якщо він $\leq \pi$, і *ввігнутим* в протилежному випадку.

Залежно від класифікації вершин m і M (ввігнута, опорна, опукла) і кута α можливі всього 18 випадків. Однак всі ці випадки можна звести до восьми (які покривають усі можливі), які наведено в табл. 3 і показано на рис. 80-81.

	$\boldsymbol{\alpha}$	$\boldsymbol{m}$	$\boldsymbol{M}$
1.	Опуклий	Ввігнута	Ввігнута
2.	Опуклий	Ввігнута	Неввігнута
3.	Опуклий	Неввігнута	Опукла
4.	Опуклий	Неввігнута	Неопукла
5.	Ввігнутий	Опукла	Опукла
6.	Ввігнутий	Опукла	Неопукла
7.	Ввігнутий	Неопукла	Ввігнута
8.	Ввігнутий	Неопукла	Неввігнута

Таблиця 3.

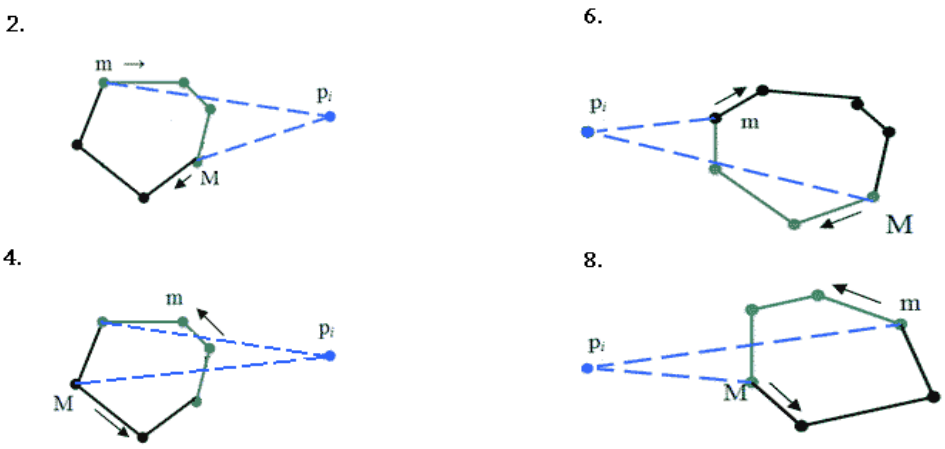


Рисунок 80. Випадки, для яких відомо, що опорні прямі існують (p_i не може бути внутрішньою точкою) і їх слід шукати в різних піддеревах дерева T

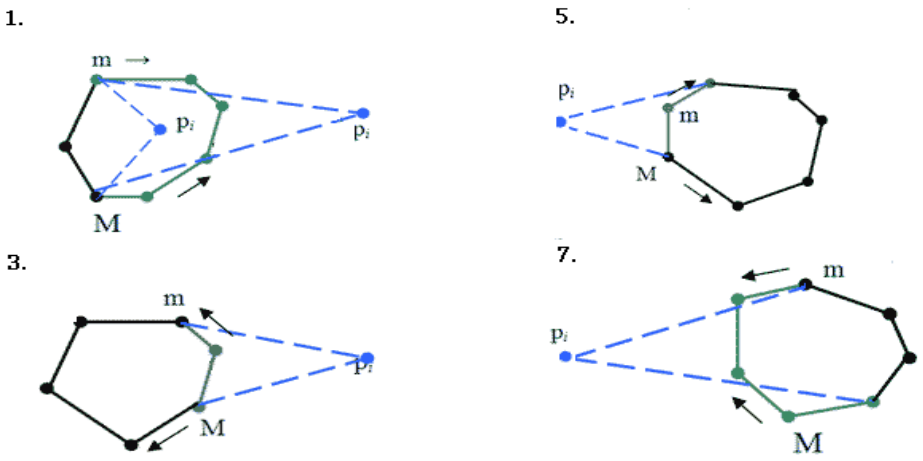


Рисунок 81. Випадки, при яких обидві опорні прямі мають шукатися в тому самому піддереві (поки не зустрінуться випадки 2, 4, 6, 8), причому у випадках 1 і 7 точка p може виявитися внутрішньою

Наступна процедура здійснює пошук вершини l (пошук r виглядатиме симетрично):

```
function ЛІВИЙ_ПОШУК(T)
  Input: дерево T, яке описує послідовність вершин
  Output: вершина l
  begin
    c := КОРІНЬ(T);
    if (pc – опорна пряма) then
      l := c
    else begin
          if (c – опукла вершина) then
          T := ЛДЕРЕВО(c) else T := ПДЕРЕВО(c)
          l := ЛІВИЙ_ПОШУК(T)
    end;
    return l;
  end.
```

Враховуючи, що дерево T збалансоване і містить не більше $i < N$ вершин, а на обробку у кожному вузлі вимагається обмежений час, оцінка пошуку $O(\log i)$.

Обидві операції РОЗЧЕПИТИ і ЗЧЕПИТИ виконуються за час $O(\log i)$. Корекція опуклої оболонки може бути виконана за час $O(\log i)$, отже маємо теорему.

Теорема 25. Опукла оболонка множини із N точок на площині може бути знайдена за допомогою відкритого алгоритму за час $\Theta(N \log N)$ з часом корекції $\Theta(\log N)$, тобто може бути побудована в реальному часі.

3.6.2 Підтримка динамічної опуклої оболонки

Метод Препарати можна розглядати як метод підтримки структури даних, що описує опуклу оболонку множини точок у випадку, коли допускається лише операція *вставки* точок.

Постає природне запитання: чи можна розробити подібну структуру даних, якщо допускається не лише вставка, але й *вилучення* точки?

Проблема: в алгоритмі Препарати точки, внутрішні для поточної опуклої оболонки, назавжди виключаються з розгляду.

Однак зараз необхідно зберігати всі точки, що містяться в множині.

Видалення деякої точки поточної опуклої оболонки може спричинити те, що деякі внутрішні точки стануть граничними точками нової опуклої оболонки (рис. 82).

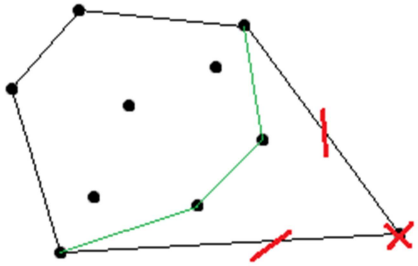


Рисунок 82. Приклад видалення точки опуклої оболонки

При вилученні точки p_3 граничними стають точки p_1 і p_2:

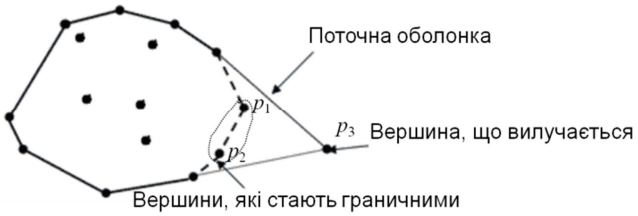


Рисунок 83. Вилучення точки p_3

Задача ОО6 (ПІДТРИМКА ОБОЛОНКИ). Задані спочатку порожня множина S і послідовність із N точок $(p_1, \dots, p_N)$, кожна з яких або додається до множини S, або вилучається з неї (очевидно, за умови, що вона належить S). Необхідно підтримувати опуклу оболонку множини S.

Ідея алгоритму – *Overmars* та *van Leeuwen*.

Буде використаний той факт, що границя опуклої оболонки є об'єднанням двох (опуклих) монотонних ламаних ліній (ланцюгів), які обмежують оболонку зверху і знизу, і відповідно називаються *B-оболонкою* і *H-оболонкою* множини точок (рис. 84).

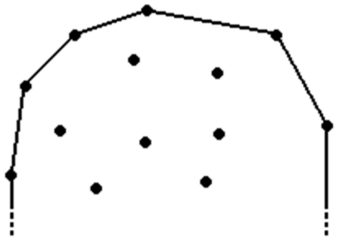


Рисунок 84. B-оболонка множини точок

Достатньо розглянути побудову однієї з цих оболонок. Зупинимося на побудові B-оболонки. Відповідна структура даних організована наступним чином.

Основа – збалансоване за висотою двійкове дерево пошуку T, листки якого використовуються для збереження точок поточної множини. Кожен проміжний вузол представляє B-оболонку точок, що зберігаються в листках відповідного піддерева.

Процедура пошуку буде проводитися відповідно до значення абсциси точок, тобто проходження листків дерева зліва направо дає множину точок, впорядковану за x-координатою.

При цьому послідовність точок B-оболонки (її вершин) також буде впорядкована за зростанням абсциси, і тому вона є підпослідовністю глобальної послідовності точок, яка зберігається в листках дерева.

Впорядковуємо S за x-координатою.

Нехай v – вузол дерева T. Позначимо через ЛСИН$[v]$ і ПСИН$[v]$ його лівого і правого нащадків відповідно.

Побудуємо B-оболонку точок, які зберігаються в листках піддерева з коренем у вузлі v. Позначимо через $U(v)$ B-оболонку множини точок, які зберігаються в листках піддерева з коренем v.

Виходячи із принципу індукції, припустимо, що вже існують $U(\text{ЛСИН}[v])$ і $U(\text{ПСИН}[v])$. Далі визначаються дві опорні точки p_1 і p_2 єдиного спільного опорного відрізку для двох оболонок.

Використовується функція $\text{З}'\text{ЄДНАТИ}(U_1, U_2)$, яка дозволяє знайти опорний відрізок для двох B-оболонок U_1 та U_2.

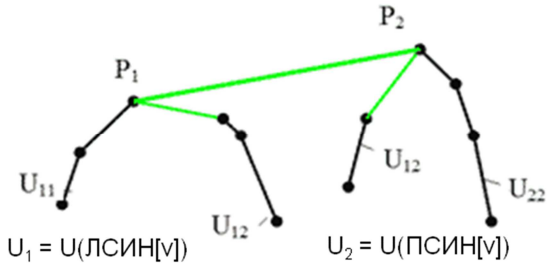


Рисунок 85. Ілюстрація функції $\text{З}'\text{ЄДНАТИ}(U_1, U_2)$

Функція $\text{З}'\text{ЄДНАТИ}$ дозволяє ефективно розчіпляти U_1 на два ланцюги, які складають впорядковану пару (U_{11}, U_{12}), й аналогічно $U_2 -$ на пару ланцюгів (U_{21}, U_{22}).

Для зберігання ланцюгів використовується зчеплена черга.

Виконується умова: опорна точка $p_1 \in U_1$ входить до U_{11}, а точка $p_2 \in U_2$ – до U_{22} (тобто в обох випадках опорна точка належить «зовнішньому» підланцюгу).

На цьому етапі, зчепивши U_{11} і U_{22}, одержуємо шукану B-оболонку $U_1 \cup U_2$ (рис. 86).

Кожен вузол v дерева T вказує на зчеплену чергу, яка представляє ту частину $U(v)$, яка не належить $U(\text{БАТЬКО}[v])$.

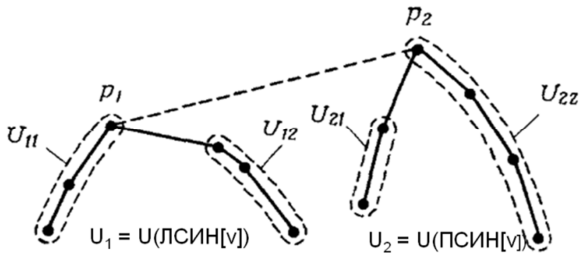


Рисунок 86. B-оболонка $U_1 \cup U_2$

Обернена операція: маючи $U(v)$, одержати $U(\text{ЛСИН}[v])$ та $U(\text{ПСИН}[v])$.

Знаючи ребро $[p_1, p_2]$, яке з'єднує опорні точки, можна розчепити $U(v)$ на ланцюги U_{11} і U_{22}, які в свою чергу можуть бути зчеплені з ланцюгами, що зберігаються в $\text{ЛСИН}[v]$ (утворення U_1) і $\text{ПСИН}[v]$ (утворення U_2) відповідно.

При цьому ребро $[p_1, p_2]$ визначається одним цілим числом $J[v]$, яке вказує на *індекс* точки p_1 у ланцюзі вершин $U(v)$.

Отже, структура даних T доповнюється наступними атрибутами, що пов'язуються з кожним вузлом v дерева:

– вказівником на зчеплену чергу $Q[v]$, яка містить частину $U(v)$, що не входить до $U(\text{БАТЬКО}[v])$ (якщо v – корінь, то $Q[v] = U(v)$, тобто в корені зберігається вся опукла оболонка);
– цілим числом $J[v]$, яке вказує на положення (індекс) лівої опорної точки в $U(v)$.

Схема функції З′ЄДНАТИ(U_1, U_2):

1. Знайти p_1 і p_2.
2. Розчепити U_1 на U_{11} та U_{12}.
3. Розчепити U_2 на U_{21} та U_{22}.
4. Зчепити U_{11} та $U_{22} \Leftrightarrow CH(S_1 \cup S_2)$: $U_{11} \cdot U_{22}$

Оцінимо пам’ять, яку використовує описана структура даних. Нехай в поточний момент множина містить N точок.

Основа структури даних – збалансоване за висотою дерево T – має N листків та $(N-1)$ вузлів на вищих рівнях. Множини точок, що зберігаються у зчеплених чергах, є розбиттям множини всіх точок. Отже, загалом використовується пам’ять $O(N)$.

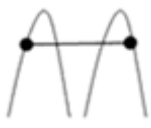
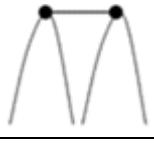
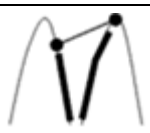
Для зчеплених черг операції розчеплення і зчеплення є стандартними, тому розглянемо операції, що виконується функцією З′ЄДНАТИ.

Лема 1. З'єднання двох розділених опуклих ланцюгів, які містять в сумі N точок, може бути виконане за $O(\log N)$ кроків.

Покажемо це. Нехай дано дві B-оболонки U_1 та U_2 і дві вершини $q_1 \in U_1$ та $q_2 \in U_2$.

Кожна із цих двох вершин може бути класифікована відносно відрізка $[q_1, q_2]$ як опукла, опорна або ввігнута. Залежно від класифікації можливі 9 випадків (табл. 4-5).

Тут $q_1 \in U_1$ та $q_2 \in U_2$. Жирним позначені ті підланцюги, які не можуть містити опорних точок, тому виключаються з подальшого розгляду.

$q_1 \mid q_2$	**ввігнута**	**опорна**	**опукла**
ввігнута			
опорна			
опукла			

Таблиця 4. Класифікація вершин q_1 та q_2 відносно відрізка $[q_1, q_2]$ як опукла, опорна або ввігнута

1.	$v(q_1)$ (або ПС $[v(q_1)]$)	ЛС $[v(q_2)]$ (або $v(q_2)$)
2.	ПС $[v(q_1)]$	ПС $[v(q_2)]$
3.	$v(q_1)$	ПС $[v(q_2)]$
4.	ЛС $[v(q_1)]$	ЛС $[v(q_2)]$
5.	Результат	Результат
6.	ЛС $[v(q_1)]$	ПС $[v(q_2)]$
7.	ЛС $[v(q_1)]$	$v(q_2)$
8.	ЛС $[v(q_1)]$	ПС $[v(q_2)]$
9.	ЛС $[v(q_1)]$	ПС $[v(q_2)]$

Таблиця 5.

Слід докладніше розглянути перший випадок (q_1, q_2) = (ввігнута, ввігнута).

Нехай пряма l_1 проходить через q_1 і її правого сусіда на U_1. Аналогічно пряма l_2 проходить через q_2 і її лівого сусіда на U_2.

Позначимо через p точку перетину прямих l_1 і l_2.

За умовою U_1 і U_2 розділимо вертикаллю l (рис. 87).

1. Нехай точка p знаходиться праворуч від прямої l. Опорна точка p_1 може належати лише зафарбованій області і $y(u) < y(v)$. Тому довільна вершина q'', що належить правому підланцюгу U_2 відносно q_2 ланцюгу, є ввігнутою відносно відрізка $[q', q'']$ (де q' – довільна вершина U_1). Тому ланцюг правіше q_2 можна не розглядати. При цьому про ланцюг ліворуч від вершини q_1 такого стверджувати не можна.

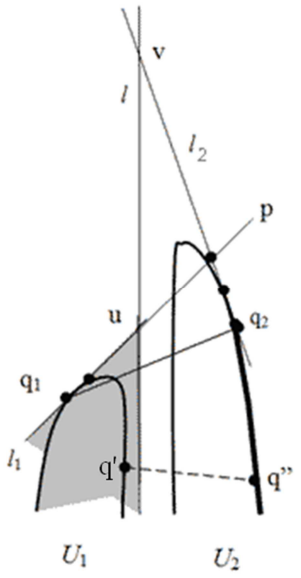


Рисунок 87. Ілюстрація першого випадку (q_1, q_2) = (ввігнута, ввігнута)

2. Аналогічно, якщо точка перетину p знаходиться ліворуч від прямої l, то підланцюг U_1 ліворуч від вершини q_1 можна не розглядати.

Нехай U_1 і U_2 – кореневі вершини дерев. Вони збалансовані, тому час виконання функції З′ЄДНАТИ(U_1, U_2) – $O(\log N)$, де N – сумарна кількість вершин в двох оболонках.

Розглянемо структуру даних T, яка описує деяке дерево оболонки. Номери точок відповідають порядку, в якому вони додавалися до множини. Кожен лист відповідає точці. Кожен нелистовий вузол – опорному відрізку. Для кожного такого вузла вказана пара $Q[v], J[v]$.

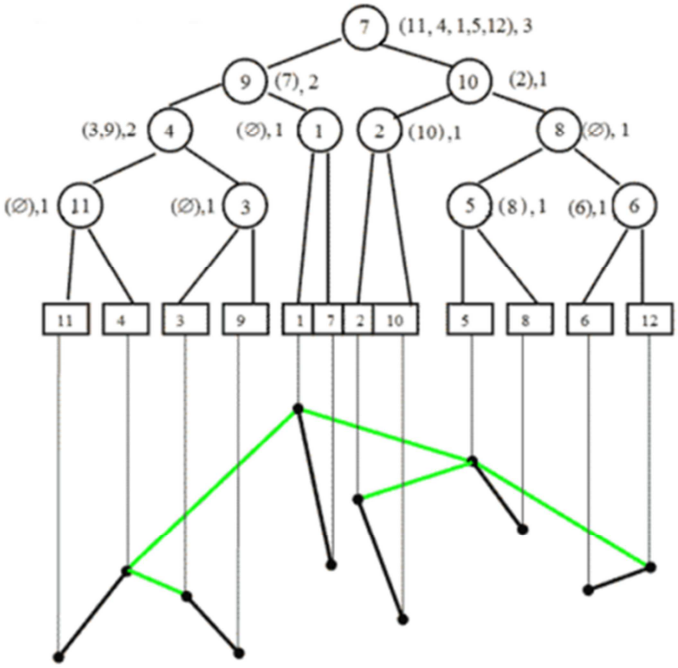


Рисунок 88. Приклад структури даних, яка описує дерево оболонки

Розглянемо вставку нової точки p. Процедура вставки повинна не лише підтримувати баланс по висоті дерева T, а й зберігати всі вимоги до структури даних T, пов’язані із зчепленою чергою.

Припустимо для простоти, що балансування не потрібне (усі дії з балансування збільшать число вузлів, які необхідно обробляти, не приводячи до будь-яких принципових відмінностей). Точка p однозначно визначає шлях із кореня дерева T до листа, в який вона повинна бути вставлена.

Рухаючись по цьому шляху із кореня, в кожному вузлі шляху ми складаємо B-оболонку, яка відноситься до цього вузла, і потім, використовуючи параметр $J[v]$, розбиваємо її на частини, щоб передати двом нащадкам цього вузла відповідні їм частини.

Вставка точки p здійснюється шляхом виклику рекурсивної процедури СПУСК(корінь$(T), p$):

procedure СПУСК(v, p)
begin if $(v \neq$ лист$)$ **then**

```
        begin
            (Q_L, Q_R) := РОЗЧЕПИТИ(U[v], J[v]);
            U[ЛСИН[v]] := ЗЧЕПИТИ(Q_L, Q[ЛСИН[v]]);
            [ПСИН[v]] := ЗЧЕПИТИ(Q[ПСИН[v]], Q_R R);
            if (x[p] ≤ x[v]) then
                v := ЛСИН[v]
            else v := ПСИН[v];
            СПУСК(v, p);
        end;
    end.
```

Спустившись, відбувається вставка нового листа і починається рух вгору.

В кожній вершині перебудовуємо оболонку її батька (розбиття $U[v] = Q_1 \cup Q_2$, $U[\text{БРАТ}[v]] = Q_3 \cup Q_4$).

```
procedure ПІДЙОМ(v)
    begin if (v ≠ корінь) then
        begin
            (Q_1, Q_2, Q_3, Q_4, J) := З'ЄДНАТИ(U[v], U[БРАТ[v]]);
            Q [ЛСИН[БАТЬКО[v]]] := Q_2;
            Q [ПСИН[БАТЬКО[v]]] := Q_3;
            U[БАТЬКО[v]] := ЗЧЕПИТИ(Q_1, Q_4);
            J[БАТЬКО[v]] := J;
            ПІДЙОМ(БАТЬКО[v]);
        end;
    else Q[v]: = U[v];
end.
```

Оцінимо складність алгоритму.

Нехай k – розмір черги до розчеплення або після зчеплення. Оскільки $k \leq N$, то час обробки одного вузла в процедурі СПУСК дорівнює $O(\log N)$. Глибина дерева T також має порядок $O(\log N)$, отже час виконання процедури СПУСК дорівнює $O(\log N^2)$ у найгіршому випадку. Процедура ПІДЙОМ дає час $O(\log N)$.

Теорема 26. Динамічна підтримка B-оболонки та H-оболонки з N точок на площині може бути виконана з часовими витратами на операції вставки і вилучення, що дорівнюють $O(\log N^2)$ в найгіршому випадку.

Якби ми використовували цей метод за умови, коли допускається лише вставка точок, тоді б часові затрати складали $O(N \log N^2)$.

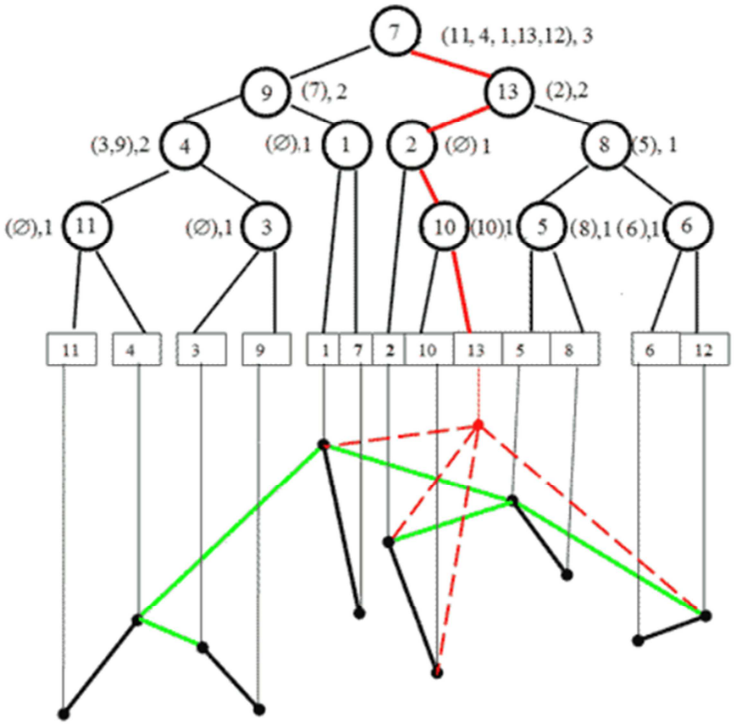


Рисунок 89. Приклад вставки точки p_{13} у структуру даних T (при вставці зачеплені вузли $\{(11,4,1,13,12),3\},\{(2),2\},\{(\emptyset),1\},\{(5),1\},\{(10),1\},\{13\}$).

3.7 Побудова опуклої оболонки методом Чана

Розглянемо ще один підхід до побудови опуклої оболонки – алгоритм Чана. Метод об'єднує два підходи: Грехема і Джарвіса, і завжди працює не гірше за них.

Час його роботи $O(n \log h)$, де h – кількість ребер опуклої оболонки. Кращої швидкості вже не досягти. Хоч алгоритм не дає принципового прискорення для методу Грехема для практичної ваги, сам підхід вартий уваги. Це швидкий алгоритм, в основі якого лежить комбінація двох повільніших.

В основі методу – «знання» остаточної кількості вершин опуклої оболонки. При цьому час на «вгадування» асимптотично такий самий, як і остаточна робота алгоритму!

Щоб досягти остаточного часу $O(n \log h)$, алгоритм Грехема має працювати за $O(h \log h)$ – кількість точок не більша за h.

Насправді, підійде будь-яке h^c, де c – константа, бо $\log(hc) = c \log h = O(\log h)$. Зокрема, $O(\log h) = O(\log h^2)$.

Множина точок розбивається на підмножини розміром h кожна, і за Грехемом будується n/h міні-оболонок: загалом час $O(n \log h)$.

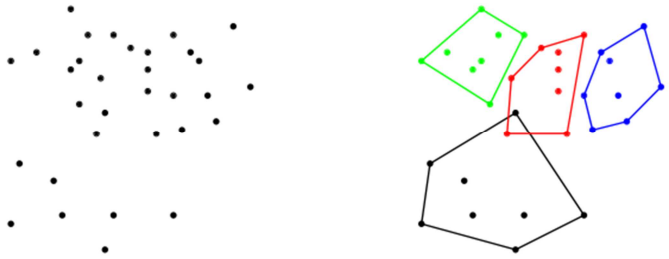


Рисунок 90. Розбиття множини точок на підмножини розміром h

До отриманих міні-оболонок застосовується обхід Джарвіса з метою отримання єдиної оболонки. При цьому міні-оболонки розглядаються як «великі точки». На кожному кроці шукаються опорні прямі з поточної вершини до кожної з міні-оболонок (включно з тією, що містить цю точку).

Кожна поточна вершина завжди належатиме загальній опуклій оболонці (і не може знаходитись всередині міні-оболонки). Серед усіх опорних прямих вибирається та, що має найменший зовнішній кут.

З однієї міні-оболонки до результуючої опуклої оболонки може увійти більше однієї вершини. Для міні-оболонок обчислюємо тільки праву опорну пряму. Якщо вершини міні-оболонок циклічно впорядковані, на пошук опорної прямої до кожної з них достатньо часу $O(\log h)$ при застосуванні бінарного пошуку.

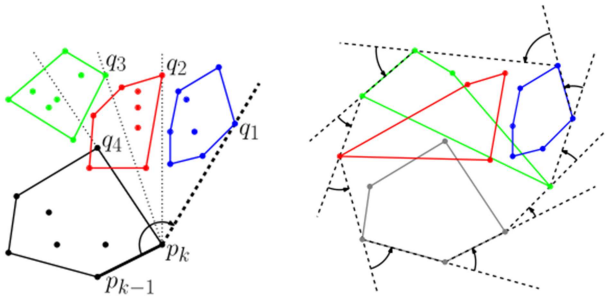


Рисунок 91. Застосування обходу Джарвіса до міні-оболонок та побудова єдиної оболонки

Оскільки h невідоме, а значення h^* є припущенням, співвідношення $h \leq h^* \leq h^2$ може не справдитись. Зокрема, виникне ситуація $h^* < h$. Про це стане відомо, коли при обході Джарвіса в оболонці виявиться точок більше, ніж h^*. Тоді процедура одразу переривається і повідомляється про помилку.

Якщо побудувалася оболонка з h^* чи менше точок, повертається результат. На фазу обходу Джарвіса, як і на обхід Грехема, пішло $O(n \log h^*)$ часу – він же загальний час роботи алгоритму.

У припущенні $h^* \leq h^2$, незалежно від успіху чи невдачі алгоритму, отримуємо оцінку $O(n \log h^*)$.

Спроба отримати єдину оболонку за $\leq h^*$ кроків.

procedure $RestrictedHull(P, h^*)$:
  $r \leftarrow [n/h^*]$;
  Розбити P на підмножини $P_1, P_2, \dots, P_r$, розміром $\leq h^*$;
  for $(i \leftarrow 1)$ **to** r **do**
    Знайти методом Грехема впорядковану $Hull(P_i)$;
  $p_0 \leftarrow (-\infty, 0)$;
  $p_1 \leftarrow$ найправіша з найнижчих точка P;
  for $(k \leftarrow 1)$ **to** h^* **do**
    for $(i \leftarrow 1)$ **to** r **do**
      знайти опорну точку $q_i \in Hull(P_i)$, тобто вершину в $Hull(P_i)$,
      щоб кут $p_{k-1}p_kq_i$ був максимальним;
      $p_{k+1} \leftarrow$ точка $q \in \{q_1, \dots, q_r\}$, що максимізує $\angle p_{k-1}p_kq$;
      if $(p_{k+1} == p_1)$ **then**
        return $\{p_1, \dots, p_k\}$ (***success***); //Після h^* ітерацій оболонка не знайдена
  return «Помилка: значення h^* замале» (***fail***);

Яким чином вибирати пробні h^*, щоб могло виконатись $h \leq h^* \leq h^2$? Маємо поступово збільшувати значення h^*, щоб не перевищити h^2. Шукана послідовність наближень $h^* = 2, 4, 16, \dots, 2^{2i}$.

Алгоритм досягне успіху, як тільки буде $h \leq h^* \leq h^2$, тобто на цьому етапі час його роботи $O(n \log h)$. Можна показати, що на всі попередні пробні етапи сумарно піде також $O(n \log h)$ часу. Отже, отримали загальну оцінку часу роботи алгоритму $O(n \log h)$.

Повний вигляд алгоритму.

procedure $Hull(P)$:
  $h^* \leftarrow 2$;
  $L \leftarrow fail$;
  while $(L == fail)$ **do**
    $h^* \leftarrow \min((h^*)^2, n)$;
    $L \leftarrow RestrictedHull(P, h^*)$;
return L.

3.8 Опукла оболонка простого многокутника

Часто розв'язок задачі вдається спростити, якщо накласти на неї певні додаткові обмеження. Множину точок можна розглядати як деякий многокутник, впорядкувавши їх певним чином і припустивши, що між сусідніми (враховуючи циклічність) точками послідовності є ребро.

Нехай відомо, що такий многокутник простий. Чи дасть нам щось ця додаткова інформація – чи можна побудувати опуклу оболонку простого многокутника за час, швидший за $O(N \log N)$?

Алгоритм Лі

Нехай p_1 – найлівіша вершина заданого простого многокутника P, а $(p_1, p_2, \dots, p_N)$ – впорядкована циклічна послідовність його вершин (за вершиною p_N іде p_1).

Нехай внутрішня частина P залишається праворуч при обході його границі у вказаному порядку (тобто множина вершин многокутника орієнтована за годинниковою стрілкою).

Нехай p_M – найправіша вершина. Вершини p_1 та p_M будуть граничними точками опуклої оболонки многокутника P. Вони розбивають послідовність вершин многокутника на два ланцюги: один від p_1 до p_M, другий – від p_M до p_1. Достатньо дослідити побудову опуклої оболонки для ланцюга $(p_1, p_2, \dots, p_M)$ – верхньої оболонки (рис. 92).

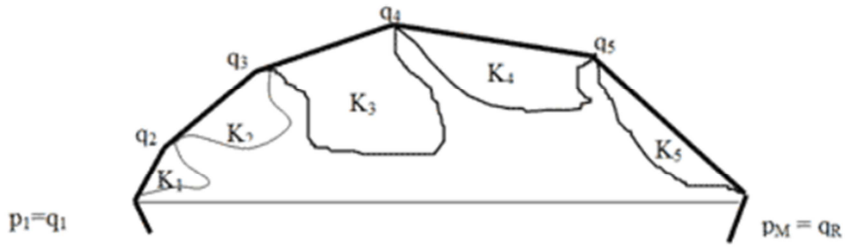


Рисунок 92. Дослідження опуклої оболонки для ланцюга $(p_1, p_2, \dots, p_M)$

Нехай $(p_1, p_2, \dots, p_R)$ – підпослідовність послідовності $(p_1, p_2, \dots, p_M)$, де $q_1 = p_1$ та $q_R = p_M$, – шукана опукла оболонка многокутника.

Кожне ребро $q_i q_{i+1}$ є «кришкою» «кишені» K_i, де кишеня K_i – підланцюг послідовності $(p_1, p_2, \dots, p_M)$, першою та останньою вершинами якого є q_i та q_{i+1} відповідно. Алгоритм проходить ланцюг $(p_1, p_2, \dots, p_M)$ та послідовно будує кришки усіх кишень.

Критичною будемо називати вершину, яка з останньою знайденою вершиною типу q утворює кишеню. Критичні вершини є кандидатами у вершини опуклої оболонки.

Крок просування – це перехід від однієї критичної вершини до іншої (рис. 93).

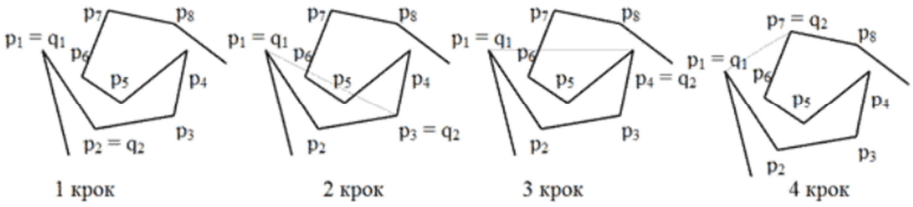


Рисунок 93. Ілюстрація просування по критичним вершинам

На третьому кроці критичною точкою є p_4. Наступною критичною точкою буде p_7. При цьому критична точка p_4 не належить опуклій оболонці.

Нехай границя многокутника переглядається від вершини p_1 до p_s $(s \leq M)$ і вершина $p_s = q_i$ є критичною. Позначимо через u вершину границі P, яка передує q_i.

Залежно від положення u відносно орієнтованого відрізка $q_M q_i$ можливі два випадки (рис. 94):

1. Вершина u знаходиться *праворуч* $q_M q_i$ або *на ньому*.

   У вертикальній смузі, визначеній вершинами q_M і q_i, досліджуються три області (1, 2, 3), які визначаються:

   - прямою, що проходить через точки q_{i-1} та q_i;
   - променем, який є продовженням відрізку $q_i u$;
   - частиною границі многокутника P, яка відповідає кишені K_{i-1}.

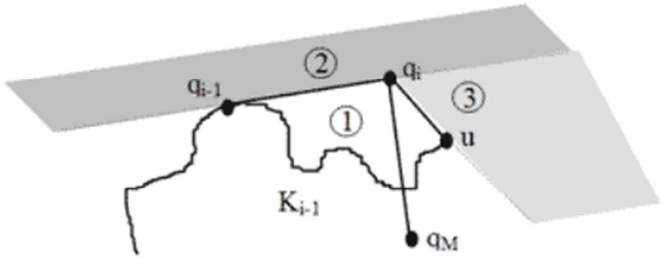


Рисунок 94.

2. Вершина u знаходиться *ліворуч* від $q_M q_i$.

   У цьому випадку до розгляду додається четверта область.

Позначимо через v вершину, яка слідує за q_i на границі многокутника P. Ця вершина може знаходитись в одній з вказаних областей розгляду. В областях 2 та 3 вершина v буде критичною, в інших – ні.

Розглянемо випадки розташування вершини v у кожній із цих областей.

Область 1. Границя многокутника заходить в кишеню.

Рухаємось по границі до тих пір, поки не досягнемо першого ребра границі, одна з вершин w якого знаходиться ззовні кишені (в області 2). Многокутник P простий, а кишеня та її кришки також утворюють прості многокутники, то, згідно з теоремою про Жорданову криву, границя многокутника P обов'язково перетинає кришку кишені (рис. 95).

Переходимо до обробки w, тобто до наступного випадку.

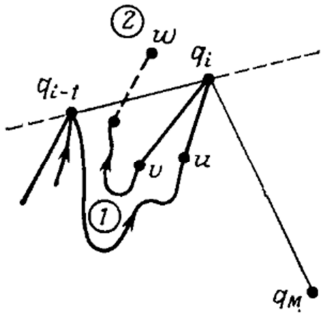


Рисунок 95. Випадок розташування вершини v в області 1

Область 2. Вершина v є критичною.

Шукається опорна пряма з вершини v до ланцюга $(q_1, q_2, \dots q_{i-1})$. Якщо пряма містить q_r $(r < i)$, то вершини $(q_{r+1}, q_{r+2}, \dots, q_i)$ вилучаються, а v береться як нова q_{r+1} (рис. 96).

v – вершина опуклої оболонки, бо вона є зовнішньою відносно поточної оболонки $(q_1, q_2, \dots, q_M)$.

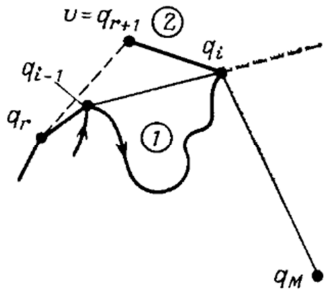


Рисунок 96. Випадок розташування вершини v в області 2

Область 3. Вершина v є критичною.

Вершина v вибирається в якості q_{i+1} (рис. 97).

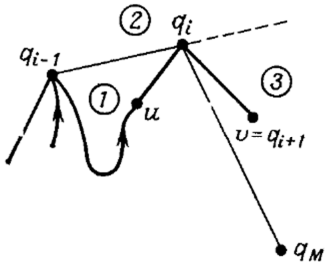


Рисунок 97. Випадок розташування вершини v в області 3

Область 4. Границя многокутника заходить всередину опуклої оболонки.

Рухаємось по границі многокутника доти, поки не досягнемо першого ребра, яке має властивість: одна з його вершин w є (а) зовнішньою до області 4 або (б) співпадає з q_M (рис. 98).

(а) вершина w належить області 3 або 2 і обробляється відповідно.

(б) процедура завершується.

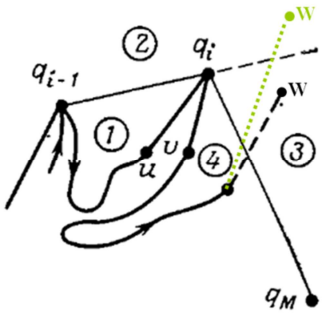


Рисунок 98. Випадок розташування вершини v в області 4

Алгоритм

Введемо позначення:

- P – черга, що містить послідовність p_i;
- Q – стек, що містить послідовність q_i;
- q_0 – фіктивна вершина: $x(q_0) = x(p_1) = x(q_1),\ y(q_0) < y(q_1)$;
- u – вершина многокутника P, що прямо передує q_i;
- v – поточна вершина многокутника P;
- $x \Leftarrow U$: виштовхнути x з U;
- $U \Leftarrow x$: заштовхнути x в U.

```
procedure ОО_ПРОСТОГО_МНОГОКУТНИКА(p_1, ..., p_M)
begin
   P ⇐ (p_2, ..., p_M);
   Q ⇐ q_0;
   Q ⇐ p_1;
   while (P ≠ ∅) do
      begin
         v ⇐ P;
         if ((q_{i-1} q_i v) – правий поворот) then          (*області 1, 3, 4*)
            if ((u q_i v) – правий поворот) then            (*області 3, 4*)
               if (q_M q_i v) – правий поворот) then        (*область 3*)
                  Q ⇐ v
               else                                         (*область 4*)
```

```
            while (ПЕРЕДНІЙ(P) знаходиться зліва від q_M q_i або на ньому) do
              ВИШТОВХНУТИ P
        else                                              (*область 1*)
            while (ПЕРЕДНІЙ(P) знаходиться зліва від q_i q_{i-1} чи на ньому) do
              ВИШТОВХНУТИ P
    else                                                  (*область 2*)
      begin
        while ((q_{i-1} q_i v) – лівий поворот) do
          ВИШТОВХНУТИ Q;
        Q ⇐ v;
      end;
  end;
end.
```

Аналіз складності алгоритму

Після ініціалізації кожна вершина границі відвідується рівно один раз, перш ніж вона буде прийнята або виштовхнута. Обробка кожної вершини многокутника здійснюється за константний час. Послідовності $(p_1, \dots, p_M)$ та $(q_1, \dots, q_R)$ містять $O(M)$ елементів.

Нижня оболонка будується аналогічно.

Теорема 27. Опукла оболонка простого многокутника з N вершинами може бути побудована за оптимальний час $\Theta(N)$ при використанні пам'яті об'ємом $\Theta(N)$.

3.9 Апроксимація опуклої оболонки

При знаходженні наближеної опуклої оболонки ми розмінюємо точність на простоту та ефективність алгоритму. Подібний алгоритм може бути корисним для задач, в яких необхідно швидко знайти розв'язок, причому достатньо, щоб він був наближений. Зокрема, у прикладній статистиці результати спостережень відомі з деякою визначеною мірою точності.

Розглянемо один з таких алгоритмів.

Основна ідея алгоритму: виділити із заданої множини точок S деяку підмножину S^*, опукла оболонка якої є апроксимацією опуклої оболонки S (рис. 99).

Зауважимо, що в даній схемі виділення апроксимуючої підмножини точок буде використана модель обчислень з примітивною операцією взяття цілої частини.

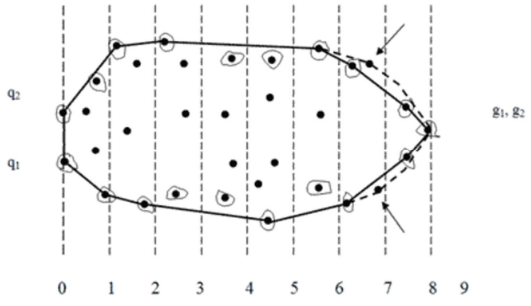


Рисунок 99. Апроксимація опуклої оболонки

Побудова

1. Визначаються чотири екстремальні точки:

$$q_{1=}\min_y \min_x S, \qquad q_2 = \max_y \min_x S,$$

$$g_1 = \min_y \max_x S, \qquad g_2 = \max_y \max_x S.$$

Беремо точки, які мають мінімальну та максимальну x-кординати: x_{max}, x_{min}. Вертикальна смуга між ними розбивається на k смуг рівної ширини. Ці k смуг утворюють послідовність «комірок», по яким будуть розподілятись N точок множини S.

2. В кожній із цих k смуг визначаються точки з екстремальними y-координатами: $S_i^* = \{P^i_{\min y}, P^i_{\max y}\}$ (всього $2k$ точок).
3. Для 1-ї та k-ї смуг екстремальних точок буде не більше чотирьох:

$$S_1^* = \{P^1_{\min y}, P^1_{\max y}, q_1, q_2\}, \qquad S_k^* = \{P^k_{\min y}, P^k_{\max y}, g_1, g_2\}.$$

Загалом сформована множина S^* містить не більше $2k + 4$ точок.

4. Одним з відомих методів (наприклад, Грехема) будується опукла оболонка множини S^*, яка є апроксимацією оболонки заданої множини S.

Це справді лише апроксимація – не всі точки істинної оболонки увійшли до побудованої (рис. 99).

Реалізація

1. Вказані k смуг відображаються в масиві із $(k + 2)$-х елементів (0-й та $(k + 1)$-й елементи містять дві точки із екстремальними значеннями x-координати: відповідно $(x_{\min}, x_{\max})$).
2. Смуга з номером i, в яку потрапляє деяка точка p, визначається співвідношенням: $i = [(x(p) - x_{\min})/\Delta]$, де $\Delta = (x_{\max} - x_{\min})/k$.
3. Мінімум та максимум в кожній смузі можна визначати паралельно.
4. Для побудови упорядкованої множини точок порівнюємо у кожній смузі значення x-координати двох точок множини S^*, які належать цій смузі.
5. Опукла оболонка будується модифікацією методу Грехема (метод Ендрю).

Повний час роботи алгоритму складає $\Theta(N + k)$.

Оцінимо точність апроксимації: як далеко від наближеної опуклої оболонки може бути точка із S, яка розташована за її межами?

Теорема 28. Довільна точка $p \in S$, яка не потрапила всередину апроксимованої опуклої оболонки, розташована від неї на відстані, що не перевищує $\Delta = (x_{\max} - x_{\min}) \,/\, k$ (рис. 100).

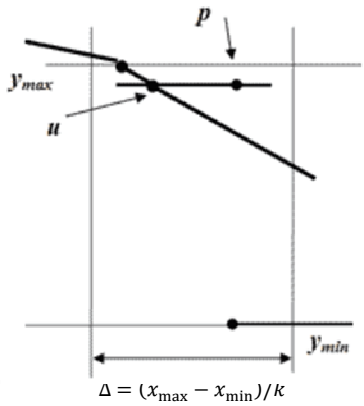


Рисунок 100. Точність апроксимації

Доведення

Розглянемо смугу, яка містить точку p. Оскільки точка p розташована за межами наближеної оболонки, то вона не може мати ні найбільшу, ні найменшу y-координату серед точок, які потрапили в цю ж смугу: $y_{\min} \le y(p) \le y_{\max}$.

Якщо u – це точка перетину горизонтальної прямої, яка проходить через точку p, з ребром наближеної оболонки, то довжина відрізка $[p, u]$ обмежує зверху відстань від точки p до оболонки. Але довжина відрізка $[p, u]$ сама обмежена зверху шириною смуги $\Delta = (x_{\max} - x_{\min})/k$.

Розглянутий метод апроксимації дає наближену оболонку, яка є *підмножиною* істинної опуклої оболонки.

Невелика зміна в алгоритмі дозволить отримати *«супермножину»* істинної оболонки.

Для кожної внутрішньої смуги достатньо кожну екстремальну точку p замінити парою точок на границі смуги з ординатою $y(p)$.

Отримана в результаті оболонка охоплюватиме істинну опуклу оболонку.

Будь-яка точка наближеної оболонки буде знаходитись на відстані не більше, ніж $(x_{\max} - x_{\min})/k$ від істинної.

На рис. 101 показана побудова «*супермножини*» істинної оболонки. Обведені всі екстремальні точки. Іншим кольором позначені пари точок, що замінюють екстремальні, та охоплююча опукла оболонка. Для порівняння зображена й істинна опукла оболонка.

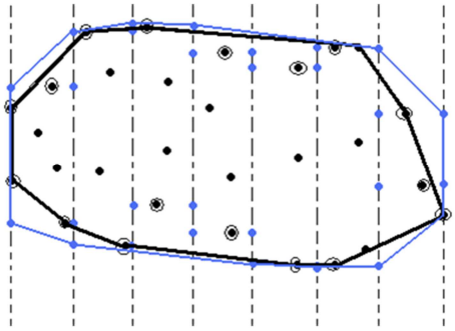


Рисунок 101. Побудова *«супермножини»* істинної оболонки

3.10 Побудова опуклої оболонки в 3D

3.10.1 Особливості представлення

Нехай в просторі E^d задано k різних точок $p_1, p_2, \dots, p_k$. Тоді множина точок $p = a_1p_1 + a_2p_2 + \cdots + a_kp_k\big(a_j \in R, a_1 + a_2 + \cdots + a_k = 1\big)$ – *афінна множина* (рис. 102а), породжена точками $p_1, p_2, \dots, p_k$, p *– афінна комбінація* $p_1, p_2, \dots, p_k$ (рис. 102б).

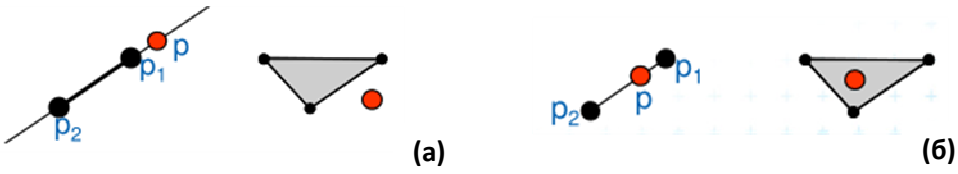


Рисунок 102. Афінна множина (а) та афінна комбінація (б)

Опукла множина і *опукла* комбінація: додаткова умова $a_j \geq 0$.

Наприклад, при $k = 2$ опукла множина – відрізок, що з'єднує дві задані точки, а афінна множина – пряма, яка проходить через ці точки.

Афінна оболонка $aff(L)$ множини L: найменший афінний простір, який містить L.

Наприклад, при $k = 3$ афінною оболонкою плоского опуклого многокутника є площина, що його містить.

Множина точок буде *афінно незалежною*, якщо жодну точку не можна представити через афінну комбінацію інших.

Опуклий d*-політоп* (політоп): скінченна d-вимірна поліедральна множина.

d*-симплекс* (симплекс): d-політоп, який є опуклою оболонкою $(d + 1)$ афінно незалежних точок.

d-політоп *симпліціарний*, якщо кожна з його гіперграней є симплексом.

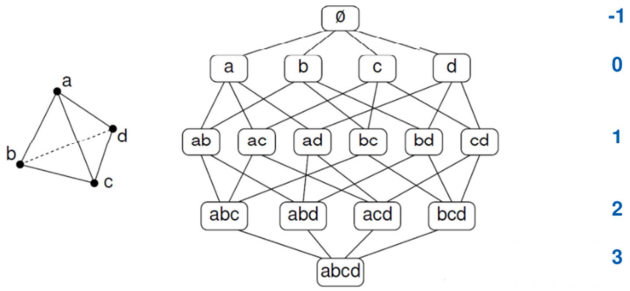


Рисунок 103. Діаграма Хассе (граф граней) для тетраедра

В загальному випадку кількість граней d-політопа має порядок $O\left(N^{d/2}\right)$.

Оцінимо при $d = 3$ залежність числа граней та ребер від кількості вершин n. Формула Ейлера пов'язує кількість вершин $V = n$, ребер E і граней F: $V - E + F = 2$.

Візьмемо тетраедр (3-симплекс) (рис. 104). Всі його вершини належать опуклій оболонці, а грані є трикутниками (тобто більше розбивати нікуди).

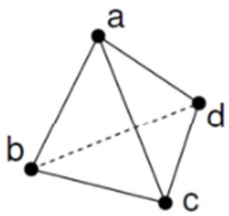


Рисунок 104. Тетраедр 3-симплекс

Кожна грань містить три ребра, кожне ребро є суміжним з двома гранями: $3F = 2E$. Підставимо у формулу Ейлера:

$$V - E + \frac{2E}{3} = 2,$$

$$V - 2 = \frac{E}{3},$$

$$E = 3V - 6 < 3V = 3n = O(n),$$

$$F = \frac{2E}{3} = 2V - 4 < 2V = 2n = O(n).$$

Теорема 29. Для d-політопа з n вершин кількості ребер і граней не перевищать відповідно $3n - 6$ та $2n - 4$, причому рівність буде у випадку симпліціарного політопа.

Тому для представлення 3-політопа досить лінійної пам'яті.

Покажемо, яким чином можна перетворити поліедр на планарний граф (рис. 105). Видно відповідність між вершинами і ребрами многогранника та його 1-скелета. Всі грані, крім однієї, відповідають замкненим областям графа; цій останній грані відповідає зовнішня необмежена область.

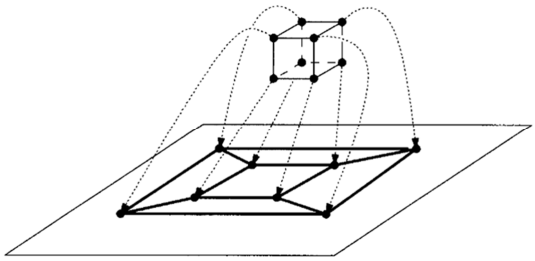


Рисунок 105. Перетворення поліедра у планарний граф

3.10.2 «Метод загортання подарунка»

Даний метод є прямим узагальненням обходу Джарвіса. Аналогічно, його складність буде $O(nF)$, де F – кількість граней опуклої оболонки. Замість опорної прямої виникає поняття опорної (гіпер)площини (рис. 106).

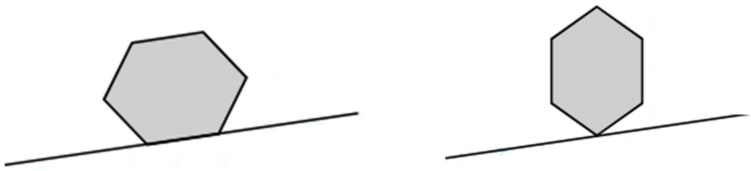


Рисунок 106. Опорна пряма та опорна гіперплощина

Припускається, що відома одна грань опуклої оболонки разом з її підгранями. Відшуковується суміжна грань відносно відомого ребра.

Площина, що містить цю (відому) грань, «загортається» по ребру в напрямку множини точок, поки не зустріне першу за них. Так визначається наступна відома трикутна грань і процес повторюється.

Більш строго, переглядаються півплощини, які мають спільну пряму, що проходить через ребро e, і серед них шукається та, що утворює найбільший опуклий кут з півплощиною, яка містить F (рис. 107).

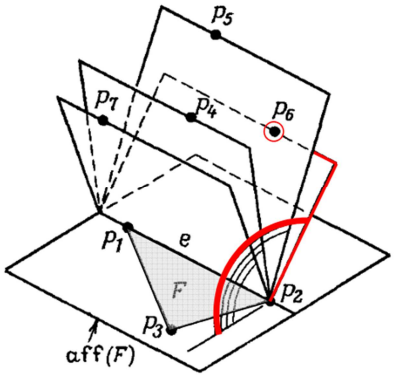


Рисунок 107. Пошук півплощини, що утворює найбільший опуклий кут з півплощиною, яка містить F

Порівняння кутів відбувається через їх котангенси. Нехай $\mathbf{n}$ – одинична нормаль до F, a – одиничний вектор, ортогональний як ребру e, так і нормалі $\mathbf{n}$ (його орієнтація аналогічна орієнтації $\mathbf{n} \times p_2p_1$).

Для кожної точки p_k обчислимо котангенс кута φ_k між півплощинами F та ep_k: $-|Up_2| \,/\, |UV|$, де $|Up_2| = \mathbf{v}_k\mathbf{a}^T$, $|UV| = \mathbf{v}_k\mathbf{n}^T$ та $\mathbf{v}_k = p_2p_1$. Шукана точка p_k має найбільше значення котангенса (рис. 108).

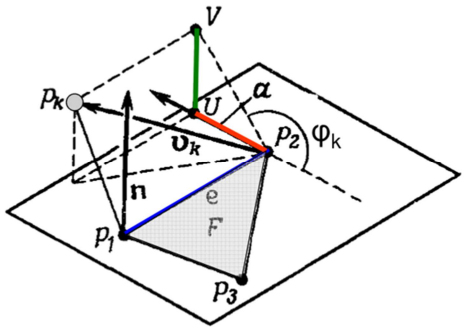


Рисунок 108. Пошук точки p_k з найбільшим значенням котангенса

Початкова грань отримується послідовною апроксимацією площини, що її містить. Будується послідовність опорних площин, кожна з яких має на одну вершину опуклої оболонки більше, ніж на попередньому кроці.

Вибирається точка p_1' з найменшою x-координатою. Вона буде вершиною (0-гранню) опуклої оболонки. Площина π_1 має бути ортогональною до вектора $(1,0,0)$ і проходити через p_1'. На кожній наступній j-й ітерації отримуємо площину π_j з нормаллю $\mathbf{n}_j$, що містить вершини $p_1', \ldots, p_j'$ та обчислюємо $\mathbf{a}_j$.

Приклад пошуку початкової площини, що містить стартову грань для алгоритму загортання подарунку. Будується послідовність площин π_1, π_2, π_3. Площина π_3 – шукана (рис. 109).

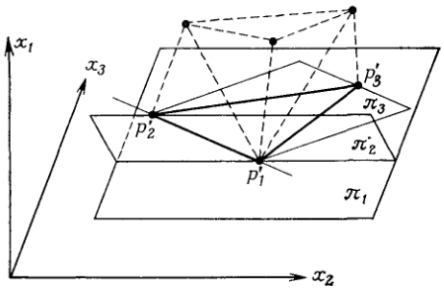


Рисунок 109. Приклад пошуку початкової площини, що містить стартову грань

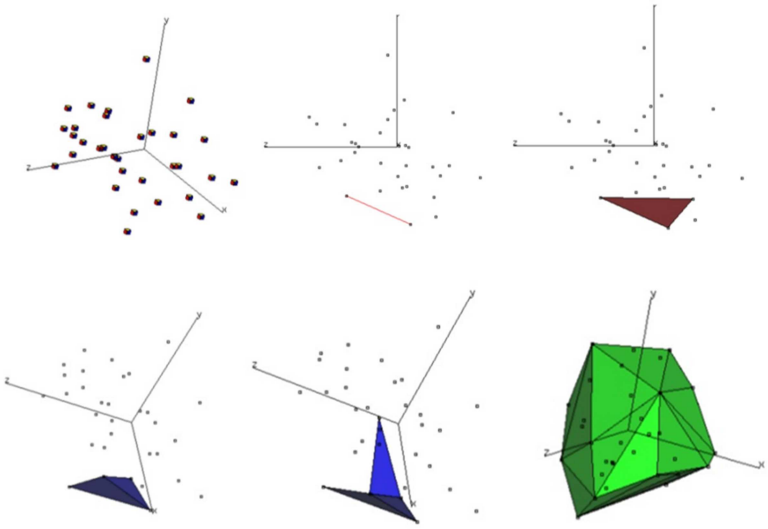


Рисунок 110. Приклад роботи «методу загортання подарунку»

3.10.3 Метод «розділяй та владарюй»

Повернемося до двовимірного випадку і розглянемо іншу версію підходу «розділяй та владарюй». Його особливість – попереднє сортування точок по x (рис. 111). На загальну оцінку часу $O(n \log n)$ воно не впливає.

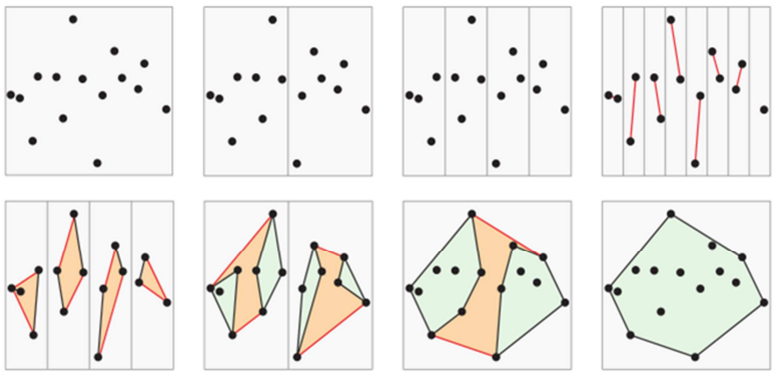


Рисунок 111. Двовимірний випадок підходу «розділяй та владарюй»

Шукаються опорні прямі двох многокутників за $O(n)$ (рис. 112). Детальніше розглянемо пошук нижньої опорної прямої (рис. 113). Беремо найправішу точку в A і найлівішу в B. Фіксуємо точку в одному многокутнику і рухаємось вниз по іншому поки не знайдеться опорна пряма до нього. Фіксуємо знайдену точку і повторюємо навпаки. І так поки не прийдемо до нижньої опорної прямої.

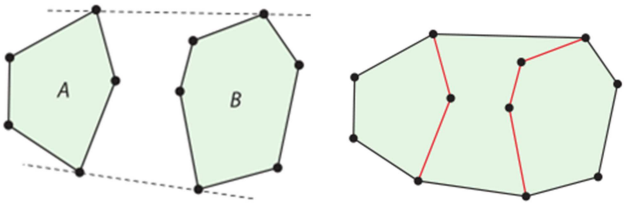


Рисунок 112. Пошук опорних прямих двох многокутників

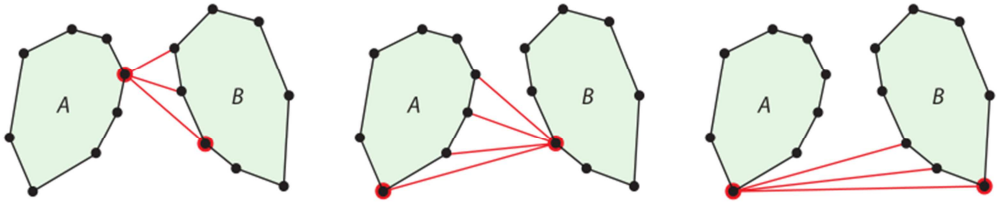


Рисунок 113. Пошук нижньої опорної прямої

Аналогічний підхід працюватиме і в 3D, причому він матиме гарантовану оптимальну складність $O(n \log n)$ (рис. 114). Точки сортуються по x-координаті. Рекурсивно виконується розбиття і будуються дві опуклі оболонки. Оболонки зливаються за лінійний час.

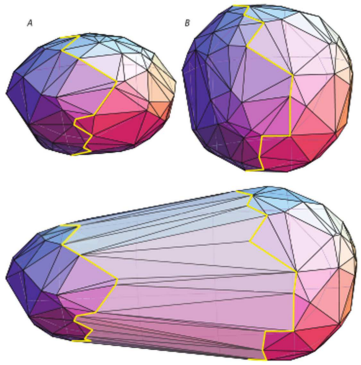


Рисунок 114. Тривимірній випадок підходу «розділяй та владарюй»

Далі визначається опорна пряма опуклих оболонок. За принципом «загортання» послідовно знаходяться трикутні грані об'єднуючого циліндра. Видаляються «невидимі» грані (рис. 115).

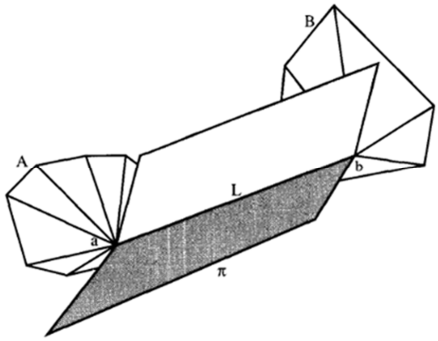


Рисунок 115. Визначення опорної прямої та пошук трикутних граней об'єднуючого циліндра

Площина π «згинається» вздовж прямої L в напрямку многогранників A і B (показані тільки грані інцидентні вершинам a та b). Кількість об'єднуючих граней буде лінійна. Кожна грань містить хоча б одне ребро одного з двох многогранників.

Отже число таких граней не перевищить сумарну кількість ребер (тобто матиме лінійний порядок). Злиття може бути проведене за лінійний час, якщо на додавання кожної грані піде константний (в середньому) час.

При «загинанні» площини ребрами-кандидатами на додавання можуть бути тільки ті, що інцидентні вершинам a та b. При цьому ребра з двох многогранників будуть додаватися завжди почергово (рис. 115).

Опорну пряму AB для двох многогранників знаходимо шляхом їх проекції на площину і застосувавши алгоритм для двовимірного випадку (рис. 116).

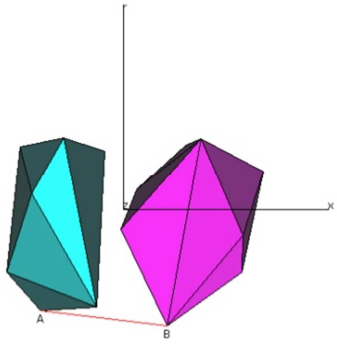


Рисунок 116. Пошук опорної прямої

Маємо знайти трикутник ABC з вершиною, що належить лівій чи правій оболонці. Тобто, це буде ребро AC лівої оболонки чи BC правої (рис. 117).

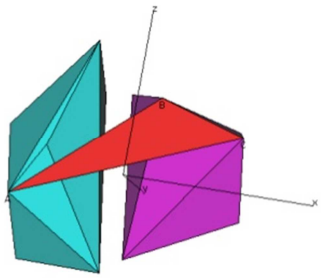


Рисунок 117. Пошук трикутника з вершиною, що належить лівій чи правій оболонці

Отримали нове ребро AC що з'єднує дві опуклі оболонки. Аналогічно знаходиться наступна грань ACD (рис. 119).

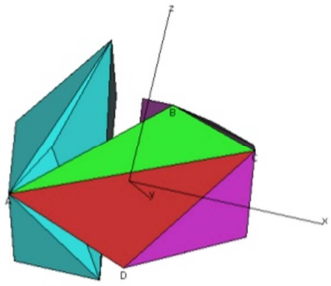


Рисунок 118. Знаходимо ребро, що з’єднує дві опуклі оболонки

Процес завершується при повторному досягненні початкового ребра AB (рис. 119).

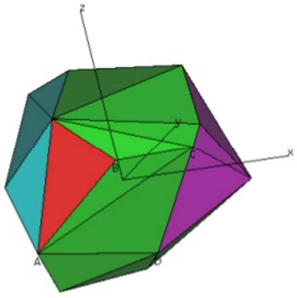


Рисунок 119. Завершуємо процес при повторному досягненні початкового ребра

Лінійний час злиття зумовлений циклічним впорядкуванням вершин на розділяючій площині H_0, яка перетинає новоутворені ребра оболонки. Для більших розмірностей такого впорядкування не існує (рис. 120).

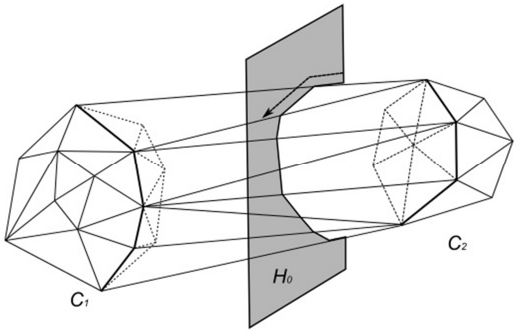


Рисунок 120.

З видаленням «невидимих» граней все насправді не настільки просто, як здається. Процедура «загортання» дає лише ребра «тіньової границі», невидимі грані потрібно шукати додатково (рис. 121). Виявляється, на самих многогранниках їх ребра тіньової границі не обов’язково утворюють простий цикл (рис. 122).

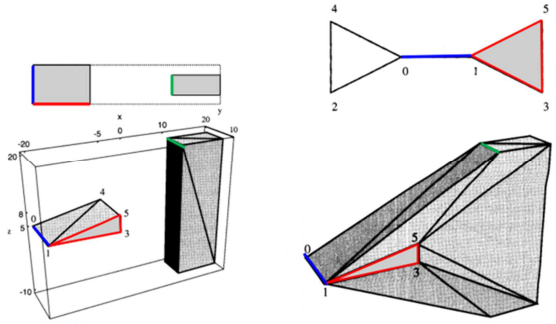


Рисунок 121. Ілюстрація процедури згортання та пошук невидимих граней

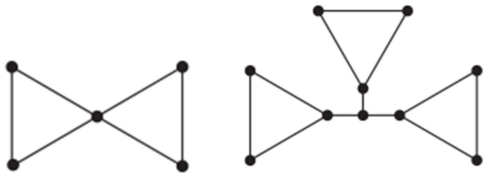


Рисунок 122. Можливі форми тіньових границь

3.10.4 Рандомізований інкрементний алгоритм

Його іноді називають методом «під-над», особливо коли йдеться про узагальнення на d розмірностей. Структура методу схожа на алгоритм Препарати: оболонка перебудовується при додаванні кожної наступної точки (рис. 123).

Серед усіх алгоритмів побудови опуклої оболонки у тривимірному просторі інкрементний метод реалізується (відносно) найпростіше, тому на практиці йому надають перевагу незважаючи на квадратичну складність.

Рисунок 123. Рандомізований інкрементний алгоритм

Створюється початковий тетраедр CH:

- беруться дві точки p_1 і p_2;
- додається третя точка поза прямою p_1p_2;
- шукається p_4, що не належить площині $p_1p_2p_3$;
- (інакше використовується 2D-алгоритм.)

Далі робиться довільна перестановка точок, що залишились. Для кожної з решти точок p_r:

- якщо p_r знаходиться всередині або на границі CH_{r-1}, то нічого не виконується: $CH_r = CH_{r-1}$;
- якщо p_r лежить ззовні від CH_{r-1}, вона додається до оболонки.

Потрібно уточнити поняття видимості. Розглянемо площину h_f, яка містить грань f (рис. 125).

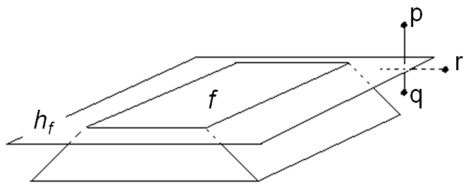


Рисунок 125. Ілюстрація видимих та невидимих граней

- Грань f *видима* з точки p, якщо вона лежить у відкритому півпросторі по зовнішній відносно многогранника бік h_f.
- Грань f *видима* з p (p лежить *над* площиною h_f).
- Грань f *невидима* з r (r лежить *на* площині h_f).
- Грань f *невидима* з q (q лежить *під* площиною h_f).

Видимі з точки p_r грані формують видимий регіон p_r на оболонці CH_{r-1}, обмежений замкненою кривою, що складається з ребер оболонки – *горизонт* p_r на CH_{r-1} (рис. 126).

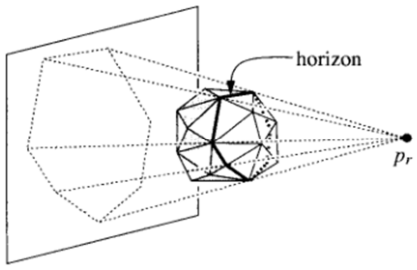


Рисунок 126. Ребра оболонки, що утворюють горизонт

Таким чином, нова опукла оболонка буде формуватися шляхом додавання нових граней – з'єднанням точки p_r з ребрами горизонту – і видалення невидимих (тобто видимих з p_r) граней.

Многогранник можна зберігати в структурі реберного списку з подвійними зв'язками, причому півребра направлені так, щоб обходили границю проти стрілки годинника, якщо дивитися на поліедр ззовні (рис. 127).

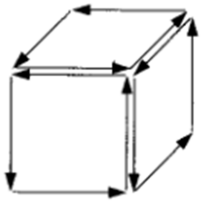


Рисунок 127. Многогранник у вигляді реберного списку з подвійними зв’язками

Тоді ми зможемо за лінійний час внести зміни в структуру при перебудові оболонки, якщо знатимемо старі і нові грані.

Окремо слід розглянути копланарний випадок для нової грані, коли точка p_r лежить в одній площині з гранню f (рис. 128). Тоді грань f є невидимою з точки і новоотримана грань має бути злита з f.

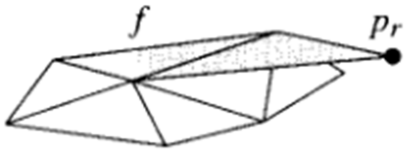


Рисунок 128. Копланарний випадок для нової грані

Зупинимося на питанні визначення граней CH_{r-1}, видимих з точки p_r.

Наївний підхід. Оскільки кожна перевірка на видимість (по який бік від площини знаходиться точка) виконується за константний час, переглянемо всі ребра. На перевірку всіх ребер піде лінійний час. Загальна складність алгоритму $O(n^2)$.

Ідея. Будемо зберігати додаткову інформацію, що допоможе визначати видимість граней. Для кожної грані f поточної оболонки зберігаємо множину $P_{conflict}(f)$ поки що необроблених точок, з яких видно f.

І навпаки, для кожної необробленої точки p_t зберігаємо множину $F_{conflict}(p_r)$ граней поточної оболонки, видимих з p_r. Такі взаємні пари p_r та f перебувають *у конфлікті*, бо не можуть одночасно входити до опуклої оболонки – при додаванні точки p_r грань f має бути видалена.

Назвемо $P_{conflict}(f)$ і $F_{conflict}(p_r)$ *списками конфліктів*. Всю цю інформацію можна зберегти в одному *графі конфліктів*.

Граф конфліктів G є дводольним графом, з множинами вершин – необроблених точок і вершин – граней поточної опуклої оболонки (рис. 129). Конфлікти показані дугами. Тобто, якщо є дуга між точкою p_r і гранню f, то f видно з точки p_r.

Множини конфліктів для точки чи грані визначаються за лінійний час. Таким чином, на кожному кроці алгоритму ми будемо знати всі конфлікти між точками, які залишилися, і гранями поточної опуклої оболонки.

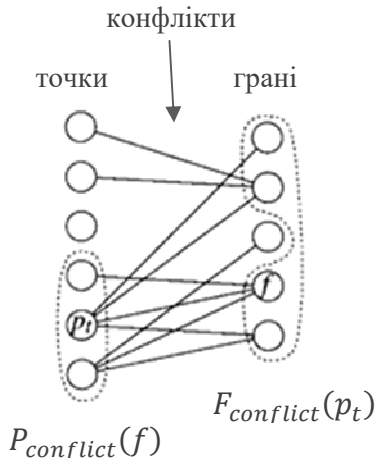


Рисунок 129. Граф конфліктів

Ініціалізація графу конфліктів G відбудеться за лінійний час (рис. 130). Проходимо по всіх точках, крім тих, які сформували початковий тетраедр CH_4, і перевіряємо видимість його граней.

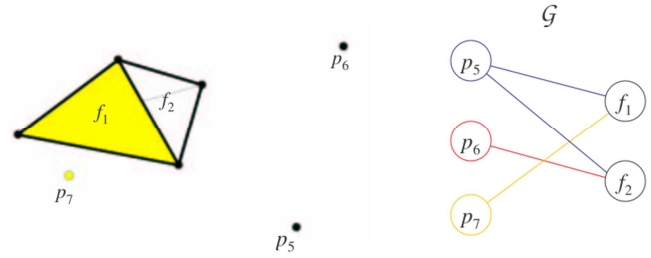


Рисунок 130. Ініціалізація графу конфліктів

Оновлення G після додавання точки p_r (рис. 131) робиться так. Спочатку вилучаються видимі з p_r грані шляхом видалення сусідів p_r в G. З графа видаляється сама точка p_r. Додаються нові грані, що з'єднують p_r з горизонтом. Визначаються нові конфлікти.

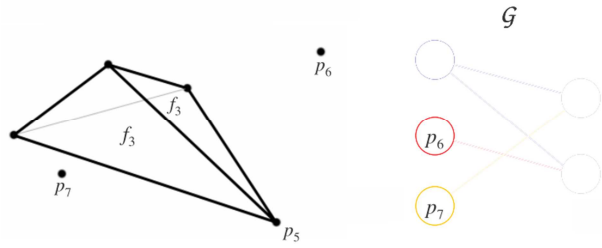


Рисунок 131. Оновлення графу після додавання нової точки

Потрібно оновити лише інформацію, що пов'язана з новими гранями, для інших граней їх множини конфліктів не зміняться.

Нехай f – нова грань оболонки CH_r. Якщо з деякої точки p_t видно грань f, то з неї видно і її ребро e. Ребро e належить горизонту p_r, тому е вже належало оболонці і було видимим з p_t в CH_{r-1} (рис. 132).

Якщо з p_t видно e, тому з неї в CH_{r-1} було видно f_1 або f_2.

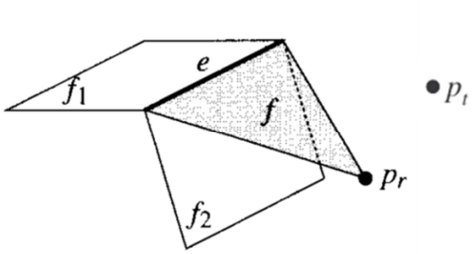


Рисунок 132.

Таким чином, p_t належало $P_{conflict}(f_1)$ або $F_{conflict}(f_2)$ в попередній оболонці CH_{r-1}. Тому для формування списку конфліктів для f достатньо перевірити точки в множинах конфліктів для f_1 і f_2 – граней, інцидентних ребру горизонту e в CH_{r-1}. У копланарному випадку список конфліктів f співпадає зі списком конфліктів f_1.

Оцінка складності методу

- Ініціалізація: $O(n \log n)$.
- Створення і видалення граней: очікуваний час $O(n)$.
- Зміни в структурі графу конфліктів G при видаленні вершини і додаванні нових граней: $O(n)$.
- Знаходження нових конфліктів: очікуваний час $O(n \log n)$.
- Загальний час: $O(n \log n)$ в середньому.
- Оцінка може досягати оптимальної, однак в гіршому випадку може доходити до $O(n^2)$.

3.10.5 Опуклі оболонки вищих розмірностей

Оскільки число гіперграней d-політопа з n вершин має порядок $\Omega(n^{d/2})$, не може існувати ефективного алгоритму побудови опуклої оболонки в розмірностях $d > 3$.

Наприклад, при $d = 4$ опукла оболонка може мати квадратичний порядок. За таких обставин алгоритм, що має складність $O(n \log n + n^{d/2})$ в гіршому випадку, можна вважати ефективним.

Зокрема, найкращий алгоритм, залежний від розміру виходу k, має час

$$O\left(n \log k + (nk)^{1-1/([d/2]+1)} \log O(n)\right).$$

На вищі розмірності узагальнюються методи «загортання подарунка», «розділяй та владарюй», рандомізований інкрементний алгоритм, *QuickHull*.

3.11 Екстремальна точка опуклого многокутника

Шукається вершина опуклого многокутника, екстремальна відносно заданого напряму. Задача виникає, зокрема, при побудові найменшого охоплюючого прямокутника зі сторонами, (наприклад) паралельними осям координат – використовуються чотири екстремальні точки.

Тривіальний алгоритм знаходить екстремум за лінійний час. Але якщо структура, в якій зберігається многокутник, допускає бінарний пошук, це можна здійснити за $O(\log n)$.

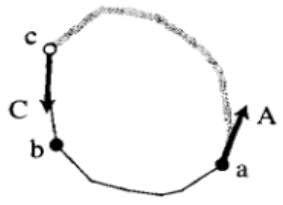
Спочатку зупинимось на пошуку найвищої точки. Для спрощення припустимо, що екстремальна точка єдина (навіть якщо там ребро, ситуацію це особливо не ускладнить).

Нехай вершини многокутника впорядковані проти стрілки годинника і екстремум має знаходитися між точками a та b (інтервал пошуку).

Основна ідея – використати направлені ребра многокутника для звуження інтервалу пошуку.

Нехай c – вершина посередині інтервалу $[a, b]$. Розглянемо можливі випадки.

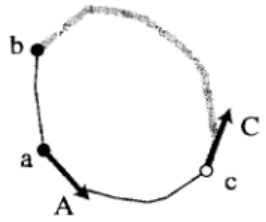
(a) Якщо ребро A, що виходить з a, направлене вгору, значить a лежить на правому ланцюгу многокутника; якщо ребро C направлене вниз, тоді c належить лівому ланцюгу. Екстремум слід шукати в інтервалі $[a, c]$ (виділений на рис. 133).



(a)

Рисунок 133. Ребро A, що виходить з a, направлене вгору

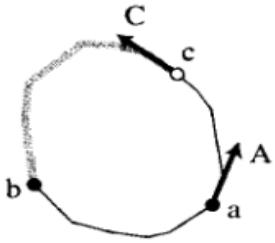
(b) Якщо A направлене вниз, а C вгору, новий інтервал пошуку $[c, b]$ (рис. 134).



(b)

Рисунок 134. Ребро A направлене вниз, а C вгору

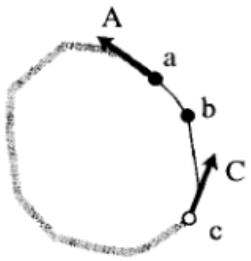
(c) Обидва ребра A і C вказують вгору, вершина a знаходиться нижче за c: новий інтервал пошуку $[c, b]$ (рис. 135).



(c)

Рисунок 135. Ребра A і C вказують вгору, а вершина a знаходиться нижче за c

(d) Обидва ребра A і C вказують вгору, вершина c знаходиться нижче за a: новий інтервал пошуку $[a, c]$ (рис. 136).



(d)

Рисунок 136. Ребра A і C вказують вгору, вершина c знаходиться нижче за a

Аналогічно розбираються випадки, коли обидва A і C направлені вниз. Алгоритм можна модифікувати, щоб знаходити екстремум відносно довільного напряму (розглядається відповідний вектор).

Перетин прямою опуклого многокутника

Алгоритм пошуку екстремуму можна використати для відповіді за час $O(\log n)$ на питання про перетин заданою прямою L опуклого многокутника P.

Нехай вектор u ортогональний до L.

Знаходимо a та b – екстремуми відносно напрямів $+u$ і $-u$. Якщо a та b лежать по один бік від L, пряма і многокутник не перетинаються. Інакше a та b розбивають фігуру на два монотонних відносно u ланцюги, перетини з якими прямої L знаходяться бінарним пошуком (рис. 137).

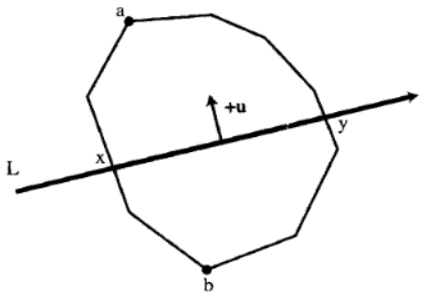


Рисунок 137. Перетин прямою опуклого многокутника

РОЗДІЛ 4
БЛИЗЬКІСТЬ ТА ПЕРЕТИНИ

4.1 Близькість. Основні алгоритми

4.1.1 Постановка і аналіз основних задач

Розглянемо задачі, пов'язані з близькістю точок на площині (задачі визначення близькості).

Задача Б1 (НАЙБЛИЖЧА ПАРА). На площині задано N точок. Знайти дві із них, відстань між якими найменша (якщо таких пар може бути декілька – достатньо знайти хоча б одну із них).

Незважаючи на просте формулювання, ця задача є однією з найважливіших в обчислювальній геометрії як з точки зору її застосувань, так і з суто теоретичної точки зору.

«Ломовий» підхід (груба сила, *brute force*) до розв'язання полягає в переборі всіх пар точок. Це дає часову складність $O(dN^2)$, де d – розмірність простору.

Чи обов'язковий повний перебір? В одновимірному випадку можна помітити, що кожна пара найближчих точок має складатись з послідовних точок множини при її упорядкуванні за значенням координати.

Таким чином, існує швидший алгоритм:

- упорядкувати N заданих дійсних чисел за час $O(N \log N)$,
- здійснити лінійний перегляд впорядкованої послідовності $(x_1, x_2, \dots, x_N)$ за $O(N)$ кроків, обчислюючи при цьому відстань $x_{i-1} - x_i, \mathrm{i} = 1, \dots, N-1$.

Цей алгоритм виявиться оптимальним.

Задача Б2 (ВСІ НАЙБЛИЖЧІ СУСІДИ). На площині задано N точок. Знайти найближчого сусіда для кожної точки множини.

Найближчий сусід – відношення на множині точок S, яке визначається таким чином: точка b є найближчим сусідом точки a (позначимо $a \to b$), якщо $dist(a,b) = \min_{c \in S-a} dist(a,c)$ (рис. 138).

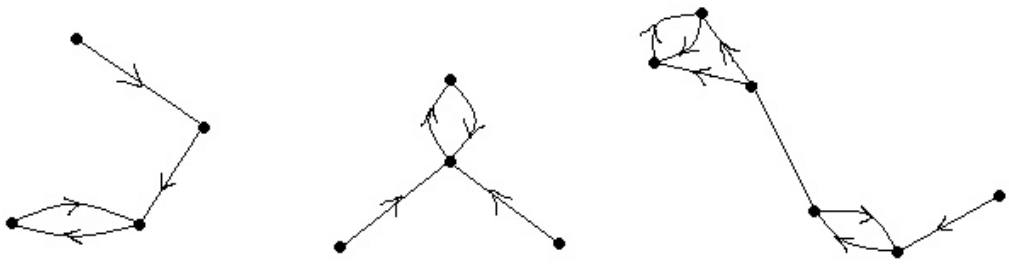


Рисунок 138. Приклади графів відношення «найближчий сусід»

Відношення «→» не є симетричним (тобто з $a \to b$ не обов'язково випливає $b \to a$) та функціональним.

Точка може мати найближчим сусідом навіть всі інші точки множини, але сама може бути найближчим сусідом лише 6-ти точок на площині та 12-ти у просторі.

Розв'язком задачі Б2 є сукупність впорядкованих пар (a, b), де $a \to b$.

Пару точок, яка задовольняє умову симетричності відношення, назвемо *взаємною парою*.

Так, наприклад, серед пар $(P_1, P_2), (P_2, P_3), (P_3, P_4), (P_4, P_3)$ взаємні пари: $P_3 \to P_4$ та $P_4 \to P_3$ (рис. 140).

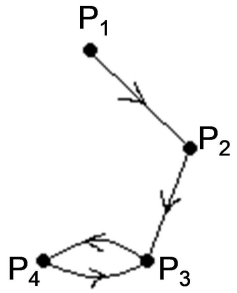


Рисунок 140. Приклад пошуку взаємної пари серед пар $(P_1, P_2), (P_2, P_3), (P_3, P_4), (P_4, P_3)$

Задача Б3 (ЕВКЛІДОВЕ МІНІМАЛЬНЕ КІСТЯКОВЕ ДЕРЕВО). На площині задано N точок. Побудувати дерево, вершинами якого є всі задані точки і сумарна довжина усіх ребер якого мінімальна (рис. 141).

Розв'язком задачі є список, який містить $N - 1$ пару точок. Кожна пара представляє ребро дерева.

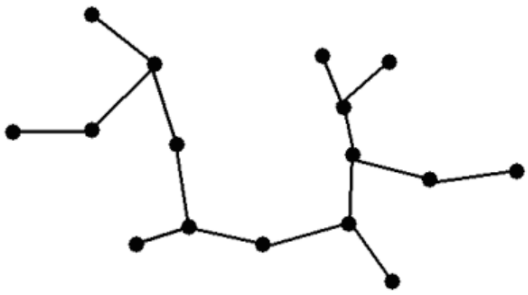


Рисунок 141. Мінімальне кістякове дерево множини точок на площині

Дерево Штейнера – дерево, яке має найкоротшу довжину за умови, що до вихідної множини дозволено додавати нові точки (рис. 142).

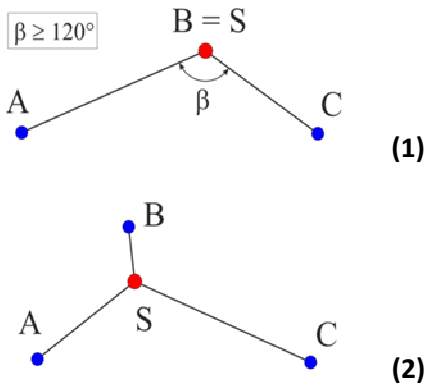


Рисунок 142. Приклади дерев Штейнера для трьох точок. Точка S додається у другому випадку

Сумарна довжина ребер дерева Штейнера може бути меншою, ніж у мінімального кістякового дерева (рис. 143):

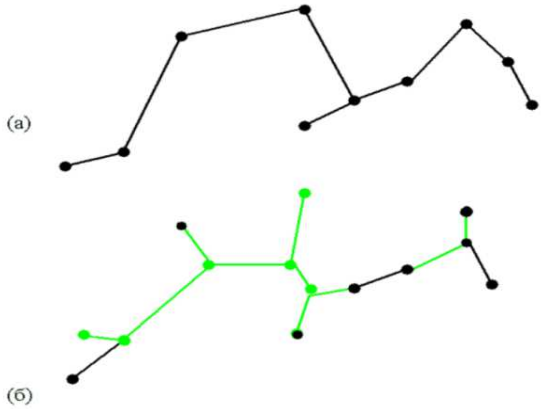


Рисунок 143. Сумарна довжина ребер дерева Штейнера (б) та кістякового дерева (а)

Побудова дерева Штейнера – NP-повна задача

Задача про мінімальне кістякове дерево в теорії графів. Задано зважений граф з N вершинами і E ребрами, необхідно знайти найкоротше піддерево графа G, яке містить усі його вершини.

Евклідова задача про мінімальне кістякове дерево. Задається N вершин на площині, що визначаються своїми $2N$ координатами, а відповідний граф містить ребра, які з'єднують кожну пару вершин.

Вага ребра дорівнює відстані між точками, з'єднаних ребром.

Така задача для довільного графа розв'язується за час $\Theta(N^2)$, однак евклідів варіант повинен мати швидше розв'язання, бо розглядається лише N вершин. Показано, що мінімальне кістякове дерево завжди містить найкоротше ребро графа.

Задача Б4 (ТРИАНГУЛЯЦІЯ). На площині задано N точок. З'єднати їх відрізками, що не перетинаються, таким чином, щоб кожна область всередині опуклої оболонки цієї множини точок була б трикутником.

Граф тріангуляції множини із N точок (рис. 144), будучи планарним, має не більше $3N - 6$ ребер. Результатом розв'язання поставленої задачі має бути як мінімум список цих ребер.

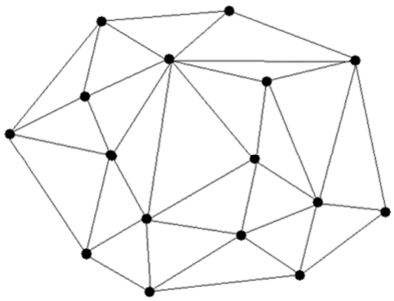


Рисунок 144. Приклад тріангуляції множини точок

Будь-яка тріангуляція множини з N точок має однакову кількість трикутників.

Для визначення «оптимальності» (виходячи зі специфіки конкретних задач) тріангуляції запропоновано багато критеріїв, зокрема:

- мінімізація повної довжини ребер,
- максимізація найменшого кута,
- критерії, що базуються на значеннях висот трикутників чи їх площі та ряд інших.

Всі розглянуті задачі (Б1–Б4) відносяться до задач, розв'язок яких шукається лише один раз і полягає в побудові деякого геометричного об'єкту.

Перейдемо до задач, що треба розв'язувати багатократно і які допускають попередню обробку даних.

Задача Б5 (НАЙБЛИЖЧИЙ СУСІД). На площині задано N точок. Як швидко можна знайти найближчого сусіда для деякої нової точки q за умови, що допускається попередня обробка?

Приклади застосування – задачі класифікації (правило найближчого сусіда), вибірка найбільш підходящих даних. Очевидний час пошуку складає $O(dN)$ в просторі розмірності d. Нас цікавитиме така попередня обробка даних, щоб пришвидшити пошук.

Невідома точка U буде класифікована як така, що належить класу B (рис. 145).

Рисунок 145. Ілюстрація задачі «найближчий сусід»

Задача Б6 (*k*-НАЙБЛИЖЧИХ СУСІДІВ). На площині задано N точок. Як швидко можна знайти k точок, найближчих до деякої нової точки q, за умови, що допускається попередня обробка?

Задача виникає при інтерполяції і виділенні контурів, а також при класифікації (правило, яке використовує k найближчих сусідів, є стійкішим, ніж те, що враховує тільки одного найближчого сусіда).

Хоч задача видається складнішою за попередню, вона розв’язується за допомогою аналогічних геометричних структур.

Задачі-прототипи

Задачі-прототипи – задачі з визначеними оцінками складності, які можна звести до тих, що розглядаються. Вони є базовими для певних класів.

Три важливі задачі-прототипи обчислювальної геометрії:

- сортування;
- визначення крайніх точок;
- унікальність елементів множини.

Всі вони мають складність $\Omega(N \log N)$. Звести ці задачі одну до іншої дуже непросто.

Так само, неможливо сформулювати природним чином задачу – їх спільного «предка».

Унікальність елементів. Задані N дійсних чисел. Визначити, чи існують серед них хоча б два рівних.

Нижня оцінка часової складності для задачі УНІКАЛЬНОСТІ ЕЛЕМЕНТІВ визначається в рамках моделі алгебраїчних дерев розв'язків.

Множину $\{x_1, \dots, x_N\}$ із N дійсних чисел можна розглядати як точку $\{x_1, \dots, x_N\}$ в E^N. Позначимо через $W \subset E^N$ множину істинності для задачі УНІКАЛЬНІСТЬ ЕЛЕМЕНТІВ на множині $\{x_1, \dots, x_N\}$ (тобто W містить усі точки, будь-яка пара координат яких різна). Можна показати, що W має $N!$ компонент зв’язності.

Твердження 1. В рамках моделі алгебраїчних дерев обчислень будь-який алгоритм, який визначає, чи є елементи множини із N дійсних чисел різними, вимагає $\Omega(N \log N)$ перевірок.

Нижні оцінки складності задач на близькість

1. Пошук найближчого сусіда

ДВІЙКОВИЙ ПОШУК $\infty_{O(1)}$ НАЙБЛИЖЧИЙ СУСІД

Нехай задано N дійсних чисел $x_1, \dots, x_N$.

У результаті двійкового пошуку визначається число x_i, найближче до числа q, яке задається в запиті на пошук. При цьому допускається попередня обробка.

Однак цю задачу можна подати і в геометричному формулюванні, поставивши у відповідність кожному числу x_i точку на площині з координатами $(x_i, 0)$. Таким чином, пошук найближчого сусіда приведе до тієї ж відповіді, що і двійковий пошук.

Терема 30. Для пошуку найближчого сусіда точки в просторі довільної розмірності необхідно виконати $\Omega(\log N)$ порівнянь в найгіршому випадку.

Якщо, залишаючись в рамках моделі дерев розв'язків, припустити, що число q рівноймовірно може належати будь-якому із $N - 1$ інтервалів, які визначаються числами x_i, то теорема дає оцінку поведінки в середньому для будь-якого алгоритму пошуку найближчого сусіда.

2. k-найближчих сусідів

НАЙБЛИЖЧИЙ СУСІД ∞ k-НАЙБЛИЖЧИХ СУСІДІВ

Зведення відразу отримаємо, поклавши $k = 1$.

Отже, наведена теорема застосовується і для задачі k-найближчих сусідів.

3. Задачі Б1–Б4

На рис. 146 подана діаграма звідності задач Б1–Б4 (символ звідності ∞ замінений стрілкою).

Рисунок 146. Діаграма звідності задач Б1–Б4

УНІКАЛЬНІСТЬ ЕЛЕМЕНТІВ ∞_N НАЙБЛИЖЧА ПАРА

Нехай задана множина дійсних чисел $\{x_1, \dots, x_N\}$. Розглядатимемо їх як точки на прямій $y = 0$, намагаючись знайти найближчу пару точок (тобто $\{x_1, \dots, x_N\}\infty_N\{(x_1, 0), \dots, (x_N, 0)\}$.

Якщо відстань між точками, які утворюють найближчу пару, не дорівнює нулю, то всі точки множини різні. Оскільки множину точок, задану в одновимірному просторі, завжди можна вкласти в простір розмірності k, то природно одержується узагальнення цього зведення.

НАЙБЛИЖЧА ПАРА ∞_N УСІ НАЙБЛИЖЧІ СУСІДИ

Одна з пар, отриманих в результаті розв'язання попередньої задачі, буде найближчою, і вона може бути визначена за допомогою $O(N)$ порівнянь.

НАЙБЛИЖЧА ПАРА ∞_N ЕВКЛІДОВЕ МКД

Евклідове мінімальне кістякове дерево містить найкоротше ребро евклідового графа на множині із N точок, тому задача НАЙБЛИЖЧА ПАРА тривіально зводиться до ЕМКД за лінійний час.

СОРТУВАННЯ ∞_N ЕВКЛІДОВЕ МКД

Маємо множину з N дійсних чисел $x_1, \dots, x_N$. Розглядаючи кожне число x_i як точку $(x_i, 0)$, побудуємо відповідну множину ЕМКД (рис. 147).

Зрозуміло, що $\{x_1, \dots, x_N\}\infty_N\{(x_1, 0), \dots, (x_N, 0)\}$.

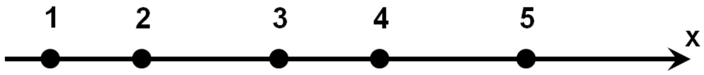


Рисунок 147. ЕМКД: $(1,2),(2,3),(3,4),(4,5) \Longrightarrow \big(O(N)\big)(1,2,3,4,\dots)$

В одержаному ЕМКД вершини, які відповідають числам x_i і x_j, з'єднані ребром $\Leftrightarrow$ коли x_i і x_j утворюють пару послідовних чисел в упорядкованій множині. Розв'язком задачі ЕМКД є список, що містить $N-1$ пару (i,j)), кожна із яких визначає ребро дерева.

Не важко перетворити цей список в упорядкований список чисел x_i, витративши на це час $O(N)$.

СОРТУВАННЯ ∞_N ТРИАНГУЛЯЦІЯ

Розглянемо множину з N точок $x_1,\dots,x_N$ розташованих так, що $N-1$ точка лежить на одній прямій, а одна розташована за межами цієї прямої (рис. 148). Маємо $\{x_1,\dots,x_N\}\infty_N\{(x_1,0),\dots,(x_{N-1},0),(x_N,-a)\}$.

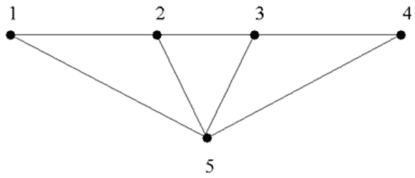


Рисунок 148. Множина N точок $(1,2,3,4,5)$

Триангуляція цієї множини може бути виконана єдиним способом.

З отриманого внаслідок триангуляції списку ребер можна одержати впорядкований список чисел x_i, витративши на це додатково $O(N)$ операцій. Таким чином, необхідно зробити $\Omega(N\log N)$ порівнянь.

Слід зауважити, що еквівалентне зведення УПОРЯДКОВАНА ОБОЛОНКА ∞_N ТРИАНГУЛЯЦІЯ базується на тому, що триангуляція множини S є планарним графом, зовнішня границя якого є опуклою оболонкою множини S.

Враховуючи, що в рамках моделі дерев обчислень обидві задачі УНІКАЛЬНІСТЬ ЕЛЕМЕНТІВ і СОРТУВАННЯ на множині із N елементів мають нижню оцінку складності $\Omega(N\log N)$, має місце наступна теорема.

Теорема 31. В рамках моделі дерев обчислень будь-який алгоритм, який розв'язує одну із задач, – НАЙБЛИЖЧА ПАРА, ЕМКД, ТРИАНГУЛЯЦІЯ, УСІ НАЙБЛИЖЧІ СУСІДИ, – вимагає $\Omega(N\log N)$ операцій.

4.1.2 Найближча пара – «розділяй та владарюй»

Нижня оцінка задачі НАЙБЛИЖЧА ПАРА має складність $\Omega(N \log N)$. Для побудови алгоритмів з такою оцінкою є два шляхи: безпосереднє використання сортування і використання методу «розділяй та владарюй».

Перший підхід одразу не підходить, бо сортування зручне лише в умовах повної впорядкованості, яка полягає в даному випадку у проектуванні всіх точок на деяку пряму, але при цьому втрачається суттєва інформація: точки p_1 і p_2 утворюють найближчу пару, але при цьому дають максимальну відстань при проектуванні на вісь y.

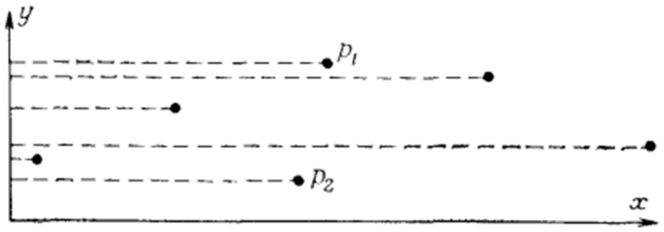


Рисунок 148. Підхід безпосереднього використання сортування (усі точки проектуються на деяку пряму)

Другий підхід полягає в розбитті задачі на дві підзадачі, розв'язок яких можна об'єднати за лінійний час, отримавши рішення вихідної задачі. Але безпосереднє застосування методу «розділяй та владарюй» в нашому випадку успіху не принесе.

Припустимо, множина розбита на дві підмножини S_1 і S_2 по $N/2$ точок і в кожній з них рекурсивно знайдено найближчу пару точок.

Однак не виключено, що найближча пара усієї множини складається з точки з множини S_1 і точки з множини S_2. А для перевірки цього потрібно ще $N^2/4$ порівнянь.

Оцінка часу роботи даного алгоритму буде $O(N^2)$, що нас не влаштовує. Тому в алгоритм потрібно внести зміни.

Одновимірний випадок

Розробимо алгоритм типу "розділяй та владарюй", що допускатиме узагальнення для двовимірного випадку.

Нехай точка m розбиває множину на дві підмножини S_1 та S_2 і при цьому $p < q$ для всіх $p \in S_1$ і $q \in S_2$.

Розв'язавши окремо рекурсивно задачу про найближчу пару для множин S_1 та S_2, отримаємо дві найближчі пари точок $\{p_1, p_2\}$ і $\{q_1, q_2\}$.

Нехай $\delta_1 = \min(S_1) = |p_2 - p_1|$ і $\delta_2 = \min(S_2) = |q_2 - q_1|$ – відстані для знайдених пар відповідно. Позначимо через δ найменшу серед знайдених δ_1 і δ_2 відстаней. Тоді найближчою парою є $\{p_1, p_2\}$ або $\{q_1, q_2\}$, або $\{p_3, q_3\}$, де $p_3 = \max(S_1)$, $q_3 = \max(S_2)$.

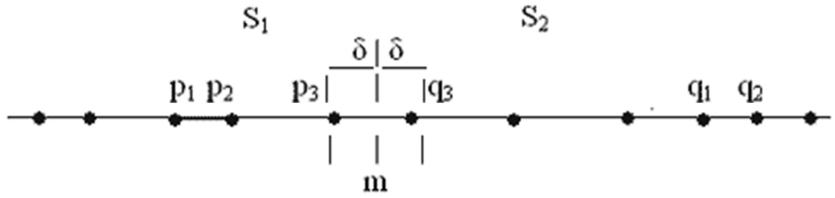


Рисунок 149. Одновимірний випадок алгоритм типу "розділяй та владарюй"

Для того щоб відстань, яку визначає пара $\{p_3, q_3\}$, була менше δ, p_3 і q_3 повинні бути на відстані, що не перевищує δ, від точки m. Відкладемо ліворуч і праворуч відносно точки m відрізки довжиною δ.

Скільки ж точок множини S_1 можуть міститись в інтервалі $(m-\delta, m]$?

Кожен напіввідкритий інтервал довжиною δ містить не більше однієї точки множини S_1, тому $(m-\delta, m]$ містить не більше однієї точки. Аналогічно для інтервалу $[m, m+\delta)$ і множини S_2.

Очевидно, всі точки, які потрапляють в інтервали $(m-\delta, m]$ та $[m, m+\delta)$, можна визначити, переглянувши множину за лінійний час.

Отже, визначивши $dist(\max(S_1), \min(S_2)) = |p_3 - q_3|$, знайдемо остаточно

$\delta^* = \min\big(\delta(S_1), \delta(S_2), dist(\max(S_1), \min(S_2))\big) = min(|p_2 - p_1|, |q_2 - q_1|, |p_3 - q_3|)$,

а значить й найближчу пару точок.

Таким чином, отримали алгоритм з часовою складністю $O(N \log N)$.

```
function НПАРА1(S)
    begin
        if (|S| = 2) then
            δ := |X[2] − X[1]|
        else
            if (|S| = 1) then
                δ := ∞
            else begin
                m := Медіана(S));
                Побудувати (S1, S2)        (* S1 = {p : p ≤ m}, S2 = {p : p > m} *);
                δ1 := НПАРА (S1);
                δ2 := НПАРА (S2 );
                p := max(S1);
                q := min(S2);
                δ := min(δ1, δ2, q − p);
            end;
        return δ;
    end.
```

Двовимірний випадок

Узагальнення виконується безпосередньо. Розіб'ємо множину точок на площині S на дві підмножини S_1 та S_2 вертикальною прямою l, що є медіаною множини S за x-

координатою так, щоб кожна точка множини S_1 лежала лівіше будь-якої точки з S_2 (рис. 150).

Розв'язавши рекурсивно задачу для S_1 та S_2, одержимо значення δ_1, δ_2 – мінімальні відстані для множин S_1 та S_2 відповідно.

Покладемо $\delta = min(\delta_1, \delta_2)$.

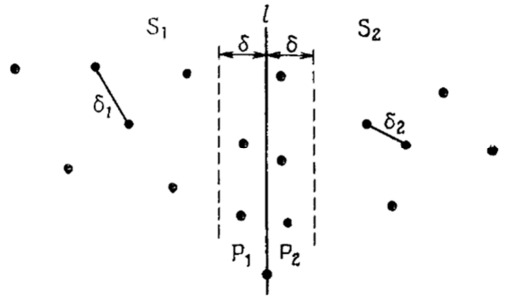


Рисунок 150. Узагальнення одновимірного випадку алгоритм типу "розділяй та владарюй" до двовимірного

Якщо найближчу пару утворюють точки $p = S_1$ і $q = S_2$, то відстань від точок q та p до l не перевищує δ.

Позначимо через P_1 і P_2 вертикальні смуги шириною δ, розташовані відповідно ліворуч та праворуч від l. Тоді $p \in P_1$ і $q \in P_2$ (рис. 151).

В одновимірному випадку було не більше одного кандидата для точок q і p, тепер же на площині таким кандидатом може бути будь-яка точка, якщо вона знаходиться на відстані, не більшій за δ від прямої l.

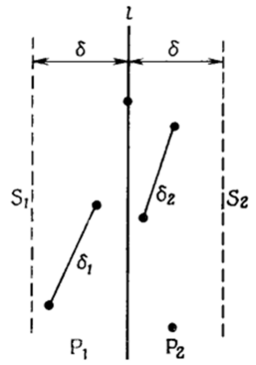


Рисунок 151. Пряма l розділяє множину точок на дві вертикальні смуги P_1 і P_2

Розглянемо в смузі P_1 довільну точку p. Необхідно знайти всі точки q із P_2, які віддалені від p не більше ніж на δ.

Виявляється, усі вони розташовуються в прямокутнику R розміром $\delta \times 2\delta$. Максимальна кількість точок, які можна помістити в такий прямокутник так, щоб відстань між ними була не менша за δ, дорівнює 6.

Це означає, що для кожної точки із P_1 слід дослідити лише не більше 6 точок із P_2.

Тому на кроці злиття розв'язків підзадач треба виконати не більше $6 \times N/2 = 3N$ порівнянь (вже не $N^2/4$).

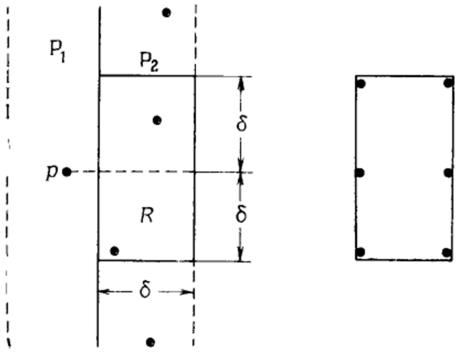


Рисунок 152. Знаходимо всі точки q із P_2, які віддалені від p не більше ніж на δ

Залишається визначити, яким чином з P_2 вибираються точки для перевірки.

Спроектуємо точку p й усі точки із P_2 на пряму l. Для визначення точок із P_2, які потрапили в R, можна розглянути лише проекції точок, що знаходяться на відстані не більшій за δ від проекції точки p.

Якщо точки впорядковані за y-координатою, то для усіх точок із P_1 кандидати на місце їх найближчого сусіда з P_2 визначаються за один прохід по впорядкованому списку.

procedure НПАРА2(S)

1. Розбити S на дві підмножини S_1 та S_2 вертикальною прямою l (медіаною).
2. Рекурсивно знайти відстань для найближчих пар δ_1 та δ_2.
3. $\delta := min(\delta_1, \delta_2)$.
4. Нехай P_1 – множина точок із S_1, які лежать в смузі на відстані δ від розділяючої прямої l, а P_2 – аналогічна підмножина в S_2. Спроектувати P_1 та P_2 на l та впорядкувати проекції за у-координатою. Нехай P_1^* та P_2^* – відповідні впорядковані послідовності.
5. «Злиття» можна виконати переглядом кожної точки з P_1^*, вивчаючи точки з P_2^*, що знаходяться на відстані, що не перевищує δ. Поки вказівник просувається по послідовності P_1^*, вказівник на P_2^* переміщується вперед-назад, залишаючись в інтервалі шириною 2δ. Нехай δ_L – мінімальна відстань між парою точок.
6. $\delta_S := min(\delta, \delta_L)$.

Позначимо $T(N)$ – час обробки алгоритмом множини із N точок. Тоді час, що пішов на обробку на кроках 1 та 5 складе $O(N)$, на кроках 3 і 6 – $O(1)$, а крок 2 потребує часу $2T(N/2)$.

Скористаємось попереднім сортуванням – впорядкуємо за ординатою всі точки. Тоді для часу обробки $P(N, 2)$, алгоритму пошуку найближчої пари, – отримаємо співвідношення:

$$P(N, 2) = 2P(N/2, 2) + O(N) = O(N \log N).$$

Теорема 32. Найкоротша відстань, яка визначається N точками на площині, може бути знайдена за час $O(NlogN)$, і цей час є оптимальним.

4.2 Задача перетину

В геоінформаційних системах використовуються різноманітні географічні карти. Вся інформація зберігається в окремих шарах (рис. 153). Кожен шар містить інформацію певного типу: адміністративні межі, дороги, житлові квартали, річки, ліси тощо. Ця інформація представлена геометричними образами, які часто можна вважати набором прямолінійних відрізків.

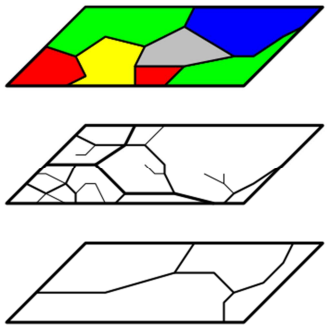


Рисунок 153. Шари географічних карт

Часто виникають задачі, пов'язані з комбінуванням (накладанням) інформації з декількох шарів (рис. 154), наприклад:

- Якою є загальна довжина лісових доріг?
- Яка загальна площа полів, що віддалені від річки щонайбільше на 1 км?

Рисунок 154. Задачі комбінування інформації з декількох шарів

Розв'язання подібних задач може звестися до пошуку перетинів множин відрізків (можливо, які є границями регіонів) (рис. 155).

Рисунок 155. Пошук перетинів множин відрізків у задачах комбінування інформації з декількох шарів

В машинній графіці одною з основних задач є видалення невидимих ліній і поверхонь, коли різні об’єкти сцени частково перекривають один одного (рис. 156). Один об’єкт перекриватиме інший, якщо їх проекції перетинаються.

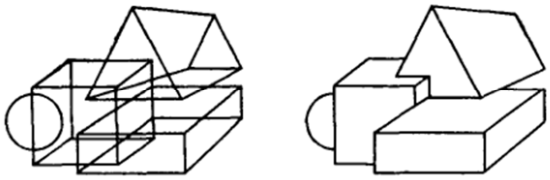


Рисунок 156. Задача видалення невидимих ліній і поверхонь різних об’єктів сцени

Важко розраховувати на ефективну реалізацію видалення невидимих ліній без використання оптимального алгоритму визначення попарних перетинів відрізків з заданого набору.

Задача перетину геометричних об’єктів також виникає в мікроелектроніці (рис. 157). Потрібно перевіряти перетин різноманітних компонент інтегральних схем, що можуть мати мільйон чи більше елементів.

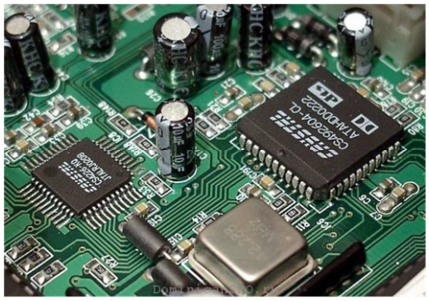


Рисунок 157. Задача перетину геометричних об’єктів в мікроелектроніці

4.2.1 Задачі перетину відрізків

Задача П1 (Перевірка перетину прямолінійних відрізків). На площині задано N прямолінійних відрізків. Потрібно визначити факт перетину хоча б двох з них.

Задача П2 (Перетин відрізків). На площині задано N прямолінійних відрізків. Треба знайти всі їх перетини (рис. 158).

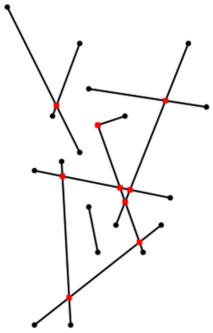


Рисунок 158. Задачі перетину

Перетин многокутників

Задача П3 (Перевірка перетину простих многокутників). Дано два простих многокутники P і Q з M та N вершинам відповідно. Треба визначити, чи перетинаються вони (рис. 159).

Оскільки P і Q прості, то при будь-якому перетині їх ребер ці ребра будуть належати до різних фігур. Якщо перетинів не знайдено, тоді $P \subset Q$ або $Q \subset P$.

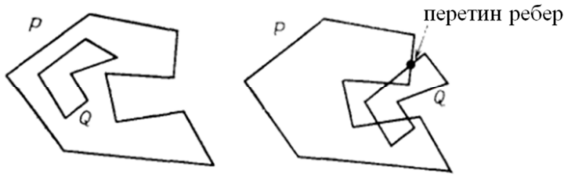


Рисунок 159. Задача перетину простих многокутників

Нехай задача П1 перевірки наявності перетинів розв'язується за час $T(K)$. Тоді факт перетину ребер P і Q буде визначений за час $T(N+M)$. Якщо перетинів не знайдено, залишається перевірити включення многокутників.

Якщо P лежить всередині Q, то всі вершини P знаходяться всередині Q. Це можна перевірити за лінійний час $O(N)$, взявши довільну вершину P. Якщо ця вершина лежить поза Q, тоді аналогічно перевіряється $Q \subset P$ за час $O(M)$.

Теорема 33. Задача *перевірки перетину простих многокутників* зводиться за лінійний час до задачі *перевірки перетину прямолінійних відрізків*.

Перевірка простоти многокутників

Задача П4 (Перевірка простоти многокутника). Дано многокутник. Потрібно визначити, чи він є простим (рис. 160).

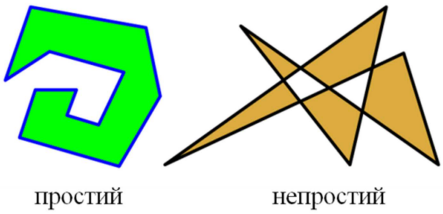


Рисунок 160. Задача перевірки простоти многкутника

Многокутник простий тоді і тільки тоді, коли жодна пара його ребер не перетинається.

Задача *перевірки простоти многокутника* зводиться за лінійний час до задачі *перевірки перетину прямолінійних відрізків*.

Задача перетину відрізків: перше наближення

Підхід грубої сили: перевірити всі можливі пари відрізків на перетин.

Час роботи $O(N^2)$ оптимальний для найгіршого випадку. Але в більшості задач очікувана кількість перетинів має набагато менший порядок. Тому бажано взяти алгоритм, що враховує також і об'єм вихідних даних (*output-sensitive*). Повний перебір однозначно поганий для перевірки простої наявності перетину.

Рисунок 161. Застосування підходу грубої сили у задачі перетину відрізків

Перетин відрізків: одновимірний випадок

Нехай на числовій осі задано N інтервалів і треба перевірити, чи не перекриваються якісь два з них. Замість перебору застосуємо сортування $2N$ точок-кінців інтервалів.

Кожну з точок можна помітити як ліву (Л) чи праву (П) відповідно до того, який край інтервалу вона представляє. Інтервали не містять перетинів тоді і тільки тоді, коли їх кінці утворюють послідовність Л, П, Л, П, …, П, Л, П (рис. 162).

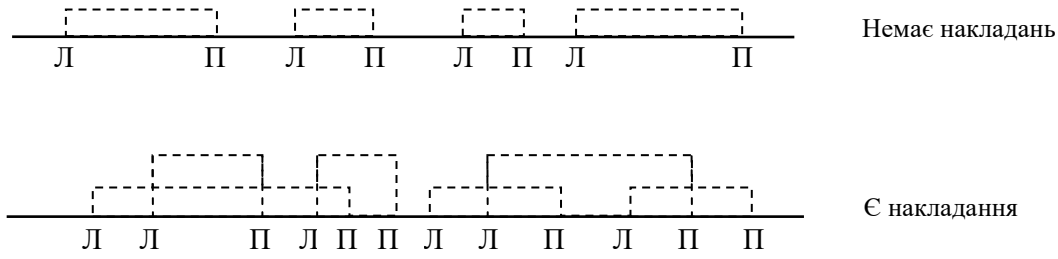


Рисунок 162. Одновимірний випадок задачі перетину відрізків

На це піде $O(N \log N)$ часу.

Знайдемо нижню оцінку алгоритму. Покажемо відповідність задачі накладання інтервалів та задачі *УНІКАЛЬНОСТІ ЕЛЕМЕНТІВ*. Раніше показали, що *УНІКАЛЬНІСТЬ N ЕЛЕМЕНТІВ* розв'язується за $O(N \log N)$ часу через сортування.

Доведемо *УНІКАЛЬНІСТЬ ЕЛЕМЕНТІВ* ∞_N *НАКЛАДАННЯМ ІНТЕРВАЛІВ.*

Заданий набір з N дійсних чисел $\{x_i\}$ можна перетворити в набір з N інтервалів $\{[x_i, x_i]\}$ за лінійний час. Ці інтервали перетинаються тоді і тільки тоді, якщо початкові числа не унікальні.

Теорема 34. Для того, що визначити, чи перекриваються N інтервалів, необхідно і достатньо $\Theta(N \log N)$ порівнянь за умови використання лише алгебраїчних функцій на вході.

Перетин двох відрізків

Розглянемо випадки перетину двох відрізків (рис. 163). Для спрощення розгляду припустимо, що вироджені випадки перетинів не зустрічатимуться:

1. Жодні три відрізки не перетинаються в одній точці.
2. Немає вертикальних відрізків
3. Два відрізки можуть перетинатися лише в одній точці (тобто немає колінеарних відрізків).

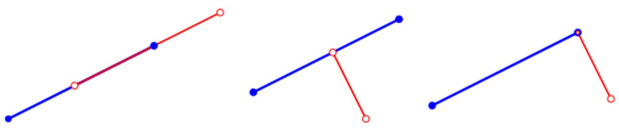


Рисунок 163. Приклади перетину двох відрізків

В реальних алгоритмах очевидно слід такі випадки обробляти. Ця обробка не вплине на асимптотичну складність алгоритму.

Два відрізки перетинаються тоді і тільки тоді, коли виконується як мінімум одна з умов (рис. 164):

1. Кожен з відрізків перетинає пряму, на якій лежить інший відрізок.

2. Кінець одного з відрізків належить іншому (граничний випадок).

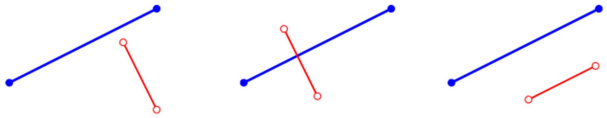


Рисунок 164. Приклади перетину та неперетину двох відрізків

Якщо треба лише констатувати наявність перетину, досить застосувати пару тестів на взаємну орієнтацію кінців (через повороти чи орієнтовану площу).

Для визначення точки перетину відрізків ab і cd використаємо їх параметричне представлення:

$$p(c) = (1-s)a + sb, \quad 0 \le s \le 1,$$
$$q(t) = (1-t)c + td, \quad 0 \le t \le 1.$$

Умова перетину: $p(s) = q(t)$. Тому шукається розвязок системи відносно s і t:

$$(1-s)a_x + sbx = (1-t)c_x + td_x,$$
$$(1-s)a_y + sby = (1-t)c_y + td_y.$$

Якщо в процесе виникає ділення на 0, то це означає, що відрізки є паралельними або колінеарними.

Розв'язання задачі перетину відрізків

Введемо порядок на відрізках (рис. 165).

s_1 та s_2 – *порівнянні в абсцисі* x, якщо існує вертикаль, що проходить через x і перетинає s_1 та s_2.

s_1 *вище* s_2 в x ($s_1 >_x s_2$), якщо вони в порівнянні в x, а точка перетину s_1 з вертикаллю x вища за точку перетину останьої з s_2.

$s_2 >_u s_4$,

$s_1 >_v s_2$,

$s_2 >_v s_4$,

$s_2 >_v s_4$,

s_3 непорівнянний.

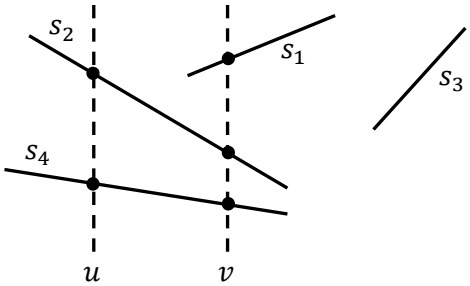


Рисунок 165. Порядок на відрізках

Відношення $>_x$ задає повне впорядкування, що змінюється в процесі руху вертикалі зліва направо.

Випадки зміни впорядкування:

1. *Зустрівся лівий кінець відрізка* s. Відрізок s додається до структури даних.
2. *Зустрівся правий кінець відрізка* s. Відрізок s видаляється зі структури даних.
3. *Зустрілася точка перетину* s_1 *та* s_2. Відрізки s_1 та s_2 обмінюються місцями в структурі даних.

Необхідна умова перетину s_1 та s_2: існує абсциса x, в якій s_1 та s_2 суміжні в порядкувані $>_x$.

Для виявлення перетинів відрізків використаємо метод *плоского замітання*. Замітання проходить зліва направо (рис. 166).

Статус замітаючої прямої буде опис відношення $>_x$ (тобто перелік вдрізків).

Списку точок подій відповідатимуть всі точки, в яких змінюється впорядкування.

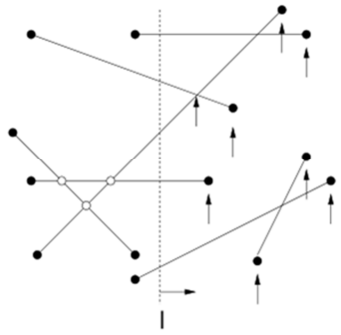


Рисунок 166. Метод плоского замітання

Структура для зберігання статусу SL має підтримувати наступні операції:

- ВСТАВИТИ (s, SL) – вставка s в повністю впорядкований статус SL.
- ВИДАЛИТИ (s, SL) – видалити відрізок s з SL.
- НАД (s, SL) – знайти ім'я відрізка – безпосереднього сусіда s згори.
- ПІД (s, SL) – знайти ім'я відрізка – безпосереднього сусіда s знизу.

Структура з такими операціями – словник. Всі вони виконуються за логарифмічний час. Якщо словник прошитий – для пошуку сусіда буде константний час.

Список точок подій змінюється динамічно, бо крім відомих точок – кінців відрізків, в процесі замітання знаходяться точки перетину, які також включаються до цього впорядкованого списку.

Перевірка на перетин проводиться для пари відрізків, що стають на цьому кроці суміжними: породження нової події x_5 при обробці події x_1. Відповідно до впорядкування, подія x_5 буде оброблена після подій x_2 , x_3 та x_4 (рис. 167).

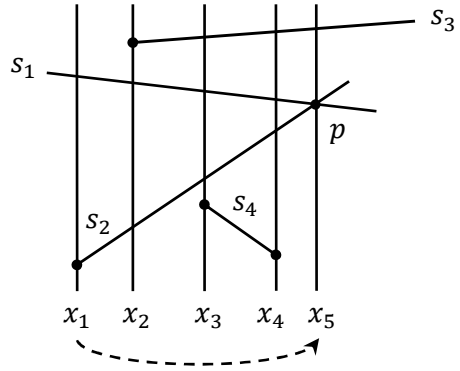


Рисунок 167.

Структура для зберігання списку точок подій Q має підтримувати такі операції:

- $MIN\ (Q)$ – знайти найменший елемент Q і видалити його.
- ВСТАВИТИ (x, Q) – вставити абсцису x у повне впорядкування в Q.
- НАЛЕЖНІСТЬ (x, Q) – визначити, чи належить абсциса x до списку точок подій Q.

Структура з такими операціями – черга з пріоритетами. Всі вони виконуються за логарифмічний час.

Алгоритм пошуку перетину відрізків:

- Площина замітається зліва направо.
- В кожній точці подій структура даних SL коректується і всі пари відрізків, що стали при цьому суміжними, перевіряються на перетин.
- Якщо деякий перетин знайшовся вперше, про нього дається звіт, і його абсциса вставляється в список точок подій Q.
- В процесі роботи алгоритм використовує допоміжний параметр A.

procedure ПЕРЕТИН_ВІДРІЗКІВ

Вхід: множина S відрізків $\{s_1, s_2 \dots, s_n\}$, що задані своїми кінцями $s = [p_a, p_b]$

Вихід: W – послідовність пар відрізків (s, s'), що перетинаються

1 Впорядкувати $2n$ кінців відрізків лексикографічно по x і y та помістити їх в пріоритетну чергу Q
2 $A \leftarrow \emptyset$
3 **while** $Q \neq \emptyset$ **do**

```
4       p ← MIN(Q)
5       if (p − лівий кінець) then
6       s ← відрізок, кінцем якого є p
7       ВСТАВИТИ(s, SL)
8       s1 ← НАД(s, SL)
9       s2 ← ПІД(s, SL)
10      if (s1 перетинає s) then A ← A ∪ (s1, s)
11      if (s2 перетинає s) then A ← A ∪ (s2, s)
12      else if (p − правий кінець) then
13          s ← відрізок, кінцем якого є p
14          s1 ← НАД(s, SL)
15          s2 ← ПІД(s, SL)
16          if (s1 перетинає s2 справа від p) then A ← A ∪ (s1, s2)
17          ВИДАЛИТИ(s, SL)
18      else {p – точка перетину}
19          (s1, s2) ← відрізки, що перетинаються в p {причому s1 = НАД(s2) зліва від p}
20          s3 ← НАД(s1, SL)
21          s4 ← ПІД(s2, SL)
22          if (s3 перетинає s2 справа від p) then A ← A ∪ (s3, s2)
23          if (s1 перетинає s4 справа від p) then A ← A ∪ (s1, s4)
24          поміняти місцями s1 та s2 в SL
25      end-if
        end-if
        {знайдені перетини потрібно занести у чергу подій Q}
26      while A ≠ ∅ do
27          (s, s′) ← A
28          x − спільна абсциса s та s′
29          if not НАЛЕЖНІСТЬ(x, Q) then
30              Додати (s, s′) у вихідну послідовність W
31              ВСТАВИТИ(x, Q)
32          end-if
33      end-do
34  end-do
```

Алгоритм перевіряє перетини тільки тих відрізків, які стали суміжними в певній точці подій. Очевидно, хибні повідомлення про перетини неможливі. Але чи не пропустить він якийсь перетин? Ні, бо якщо відрізки s_i та s_j перетинаються в точці p, тоді s_i та s_j стануть сусідами якраз перед тим, як замітаюча пряма досягне p (рис. 168).

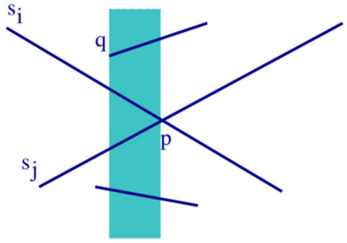


Рисунок 168.

В одній точці не може перетинатися більше 2 відрізків, тому лише s_i та s_j можуть перетинатися в точці p. Для замітаючої прямої, що знаходиться безпосередньо перед p, неможливо, щоб між s_i та s_j містився якийсь відрізок. Інакше перед p мала би бути ще інша подія.

Нехай q – подія, що передує p. Тоді порядок відрізків вздовж замітаючої прямої після події q і перед p не повинен змінитися. А отже, s_i та s_j будуть суміжними в статусі перед обробкою події p.

Оцінимо складність алгоритму. Початкове сортування $O(N \log N)$. Блоки 6-11, 13-17, 18-24 – час $O(\log N)$. За кожне виконання головного циклу **while** (рядок 3) виконається один з трьох блоків. Якщо є K перетинів, то цикл **while** виконається рівно $(2N + K)$ разів.

Загальна кількість знайдених перетинів $O(N + K)$, стільки буде перевірок в рядку 29. Кожне виконання займе $O(logN)$ операцій, тому загалом на цикл **while** піде час $O\big((N + K) \log N\big)$.

Теорема 35. На множині з N відрізків можна знайти K перетинів за час $O((N + K) \log N)$.

Перетин відрізків як факт

Нехай достатньо знайти лише один з перетинів, якщо він існує. Тоді алгоритм набагато спрощується.

```
procedure ПЕРЕВІРКА_ПЕРЕТИНУ_ВІДРІЗКІВ
begin
    сортуємо лексикографічно 2N кінців по x і y та поміщаємо їх в дісний масив
    ТОЧКА[1: 2N];
    for i := 1 until 2N do
      begin
        p := ТОЧКА[i];
        s := відрізок, кінцем якого є p;
        if (p – лівий кінец) then
            begin
                ВСТАВИТИ(s, SL);
                s1 ← НАД(s, SL);
                s2 ← ПІД(s, SL);
                if (s1перетинає s) then виводимо (s1, s)
                if (s2перетинає s) then виводимо (s2, s)
            end;
        else                                        (*p – правий кінец s*)
            begin
                s1 ← НАД(s, SL);
                s2 ← ПІД(s, SL);
                if (s1перетинає s2) then виводимо (s1, s2);
                ВИДАЛИТИ(s, SL);
            end
```

end
end.

Список точок подій міститиме лише кінці відрізків.

Теорема 36. Алгоритм ПЕРЕВІРКА_ПЕРЕТИНУ_ВІДРІЗКІВ знаходить перетин у разі його існування.

Алгоритм повідомляє про перетин, якщо він дійсно існує. Усього виконується $O(N)$ перевірок перетинів, тому деякі перетини можуть не знайтися. Головний блок **if** виконується за час $O(\log N)$ в гіршому випадку.

Теорема 37. Факт перетину довільної пари з N відрізків на площині можна визначити за оптимальний час $\Theta(N \log N)$.

Перетин відрізків

З отриманого вище результату випливає:

Наслідок 1. Наступні задачі можна розв'язати за час $O(N \log N)$ в найгіршому випадку:

- ПЕРЕВІРКА ПЕРЕТИНУ МНОГОКУТНИКІВ;
- ПЕРЕВІРКА ПРОСТОТИ МНОГОКУТНИКА;
- ПЕРЕВІРКА УКЛАДКИ ГРАФА (чи перетинаються ребра заданої прямолінійної укладки планарного графа?).

Факт перетину довільної пари з N кіл можна знайти за час $O(N \log N)$.

4.3 Діаграма Вороного

4.3.1 Історія та використання

Названа на честь нашого земляка Георгія Вороного (1868 – 1908) (рис. 169), який першим глибоко їх вивчав. Інші назви – многокутники Вороного, теселяція (мозаїка) Вороного, декомпозиція Вороного, теселяція Діріхле, многокутники Тіссена, комірки Вігнера-Зейтца, «многокутники близькості».

Це особливий вид розбиття метричного простору, що визначається відстанями до заданої дискретної множини ізольованих точок цього простору.

З кожною точкою пов'язується регіон, утворений з усіх точок, ближчих до неї, ніж до будь-якої іншої вершини – комірка Вороного (комірка Діріхле).

Рисунок 169. Вороний Георгій Феодосійович

Розглянемо групу крамниць в рівнинному місті. Нехай ми хочемо оцінити кількість споживачів певної крамниці.

За інших рівних умов розумно вважати, що споживачі обирають найближчу крамницю (рис. 170). Тоді комірку Вороного для певної крамниці (яка відтворена як точка на мапі міста) можна використати як грубу оцінку кількості потенційних клієнтів, що її відвідують.

Рисунок 170. Мапа крамниць

Можна припустити, що споживачі ходять і їздять в магазин лише дорогами, паралельними до осей координат. Тоді відстань буде обчислюватись як $d[(a_1, a_2), (b_1, b_2)] = |a_1 - b_1| + |a_2 - b_2|$.

Це – манхеттенська відстань (відстань міських кварталів), назва пов'язана з вуличним плануванням Манхеттена (рис. 171).

Рисунок 171. Відстань міських кварталів

Нехай точки представляють зерна кристала, які ростуть з постійною швидкістю в усіх напрямках. Припустимо також, що зростання зерен кристала продовжується до тих пір, поки два або більше зерен не зустрінуться. Через певний час кожне зерно, що виросло, буде представлене у вигляді комірки. У результаті буде отримана діаграма Вороного (рис. 172).

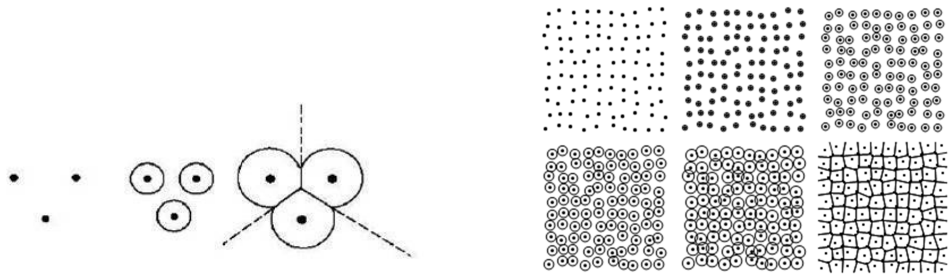


Рисунок 172. Приклад діаграми Вороного

Діаграма Вороного. Історія

Перше неформальне використання таких конструкцій можна приписати Декарту, коли він ілюстрував будову Всесвіту як заповненого ефіром і матерією та утвореного вихорами (Principia philosophiae – Засади філософії, 1644) (рис. 173).

Діріхле використовував двовимірні та тривимірні діаграми Вороного в своїй роботі про квадратичні форми (1850).

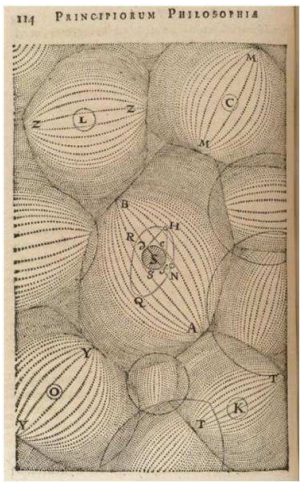


Рисунок 173. Ілюстрація будови Всесвіту Декартом

Один з засновників епідеміології Джон Сноу в 1854 році використав діаграму Вороного, щоб показати зв’язок великого спалаху холери в лондонському районі Сохо зі споживанням води з певної зараженої нечистотами колонки (на той момент ні етіологія, ні спосіб передачі хвороби не були точно відомі) (рис. 174).

Цікаво, що в поряд розташованому монастирі жоден з монахів не захворів. Але це не було аномалією. Чому? *Відповідь*: монахи пили лише пиво, яке самі ж і варили.

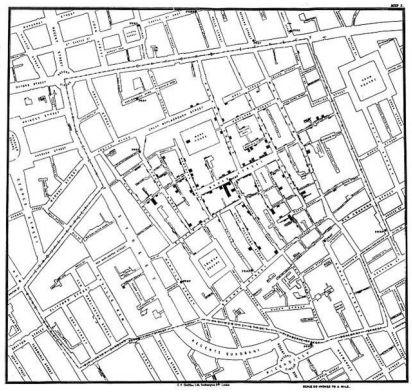


Рисунок 174. Зв’язок великого спалаху холери з нечистотами колонок

Використання діаграми Вороного

В географії (картографії) використовується при моделюванні земної поверхні, для візуалізації і аналізу видимості на місцевості, напрямів водних потоків тощо.

В археології діаграми Вороного використовуються для нанесення на карту ареалу застосування знарядь праці у стародавніх культурах і для вивчення впливу конкуренції центрів торгівлі.

Складні діаграми Вороного використовуються в матеріалознавстві, фізиці, хімії.

В екології можливості організму на виживання залежать від числа сусідів, із якими він повинен боротися за їжу та світло. Це справедливо як для тварин, так і для рослин.

В біології використання діаграми Вороного, яка відображає картину розселення тварин і розподілу життєво важливих ресурсів, допомагає досліджувати та вивчати ефект перенаселення.

В астрономії діаграми Вороного використовуються при ідентифікації груп зірок і галактик.

В кліматології діаграми Вороного служать для підрахування кількості опадів в регіоні за точковими вимірами.

Для оптимальної розбивки території (наприклад, в маркетингу чи при розміщенні небезпечних об’єктів).

В комп’ютерній графіці, задачах розпізнавання та аналізу текстів.

Робототехніка: рух роботів оминаючи перешкоди. Діаграми Вороного можливо будувати і для неточкових об‘єктів (рис. 175).

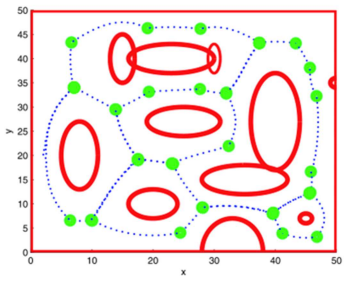


Рисунок 175. Приклад руху роботів оминаючи перешкоди

Вивчення та дослідження клітинних структур живої матерії. Приклад двовимірної моделі структури губчастої речовини кістки. Точками є центри пор (рис. 176).

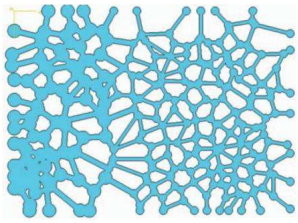


Рисунок 176. Приклад двовимірної моделі структури губчастої речовини кістки

Діаграма Вороного в архітектурі, дизайні та просто мистецтві (рис. 177-178), а також у живій природі (рис. 179).

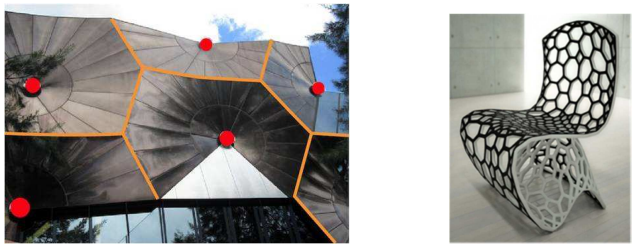


Рисунок 177. Використання діаграм Вороного в архітектурі та дизайні

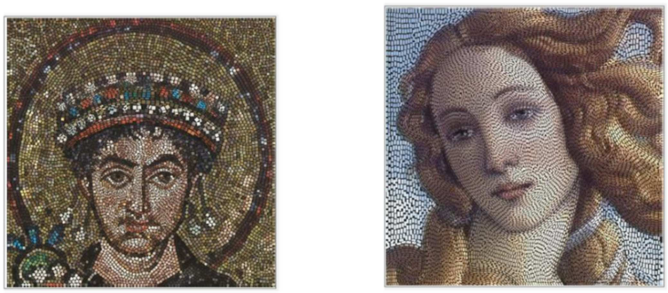


Рисунок 178. Напівавтоматичний дизайн мозаїки по зображенню

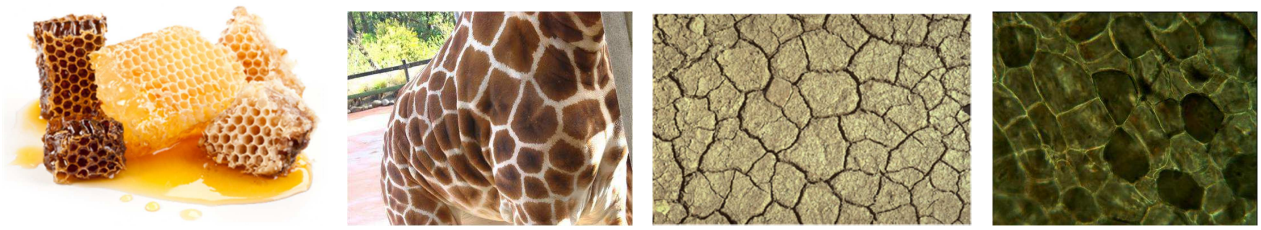


Рисунок 179. Діаграма Вороного в природі

4.3.2 Означення та властивості діаграми Вороного

Задача Б7 (ОБЛАСТІ БЛИЗЬКОСТІ). На площині задана множина S, яка містить N точок. Необхідно для кожної точки p_i множини S визначити локус (область) точок (x, y) на площині, для яких відстань до p_i менша, ніж до будь-якої іншої точки множини S (рис. 180).

Для будь-яких двох точок p_i та p_j множина точок, що є ближчою до p_i ніж до p_j, є півплощиною $H(p_i, p_j)$, яка визначається серединним перпендикуляром до відрізка $p_i p_j$ та містить саму p_i. Аналогічно визначається $H(p_i, p_j)$.

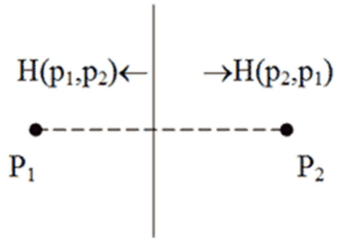


Рисунок 180. Задача області близькості

Позначимо $V(i)$ – множина точок, ближчих до p_i, ніж до іншої довільної точки. Вона одержується в результаті перетину $N-1$ півплощини. Ця множина є опуклим многокутником, який має не більш ніж $N-1$ сторону.

Область $V(i)$ – *многокутник Вороного*, що відповідає точці p_i (рис. 181). Отримані таким чином N областей утворюють розбиття площини – діаграму Вороного $Vor(S)$.

$$V(i) = \bigcap_{i \neq j} H(p_i, p_j).$$

Кожна з N вихідних точок множини належить лише одному многокутнику Вороного. Тому якщо $(x, y) \in V(i)$, то p_i є найближчим сусідом точки (x, y).

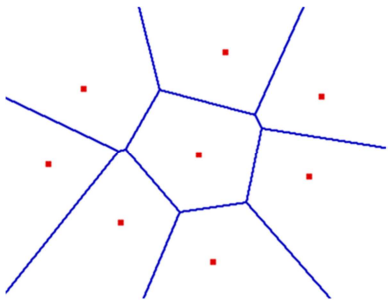


Рисунок 181. Многокутник Вороного

Коли чотири точки лежать на одному колі, то діаграма Вороного стає виродженою (рис. 182-183).

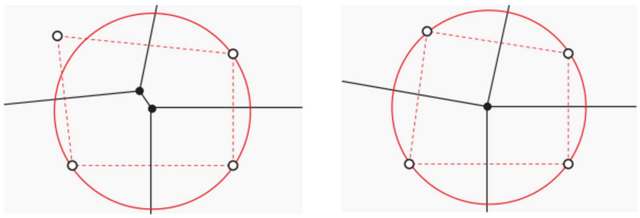


Рисунок 182. Загальний вигляд діаграми Вороного та вироджений випадок

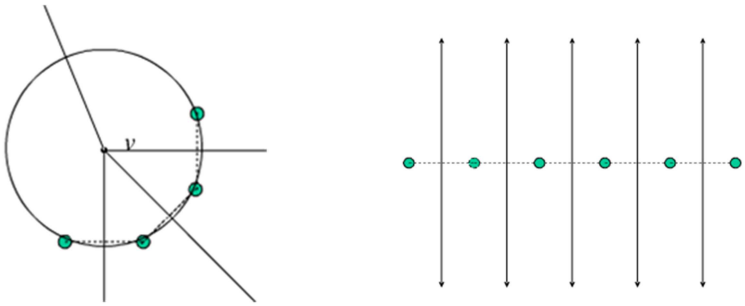


Рисунок 183. Вироджені випадки діаграми Вороного

Властивості діаграми Вороного

Припущення 1. Жодні чотири точки вихідної множини S не лежать на одному колі.

Теорема 38. Кожна вершина діаграми Вороного є точкою перетину рівно трьох ребер діаграми (рис. 184).

Нехай v є точкою перетину ребер $e_1, e_2, \dots, e_k$, перелічених за годинниковою стрілкою.

Ребро e_i є спільним для многокутників $V(i-1)$ та $V(i), i = 2, \dots, k$, а ребро e_1 є спільним для $V(k)$ та $V(1)$. Оскільки v належить ребру e_i, то вона однаково віддалена від точок p_{i-1} та p_i. Аналогічні міркування щодо рівновіддаленості v від точок p_i та p_{i+1} і т.д.

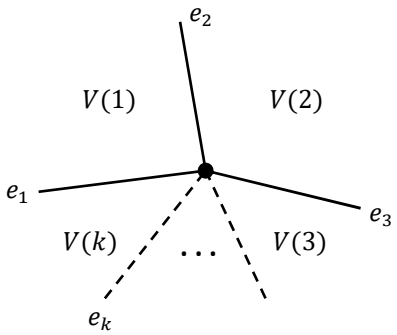


Рисунок 184. Вершина діаграми Вороного є точкою перетину трьох ребер

Отже, v рівновіддалена від точок $p_1, p_2, \dots, p_k$, тобто вони лежать на одному колі, що суперечить прийнятому припущенню при $k \geq 4$.

Тоді має бути $k \leq 3$.

При $k = 2$ обидва ребра e_1 та e_2 належать многокутникам $V(2)$ та $V(1)$, оскільки належать серединному перпендикуляру відносно p_1p_2. Але v не буде точкою їх перетину, звідки знову отримуємо протиріччя.

Тому залишається $k = 3$, і умова теореми виконується.

Наведена вище теорема еквівалентна твердженню: вершини діаграми Вороного є центрами кіл, кожне з яких визначається трьома точками вихідної множини, а сама діаграма Вороного є регулярною (всі вершини мають однаковий ступінь) зі ступенем вершин 3.

Позначимо через $C(v)$ коло, що відповідає вершині v.

Теорема 39. Для будь-якої вершини v діаграми Вороного множини S коло $C(v)$ не містить жодних інших вершин з S (рис. 185).

Нехай p_1, p_2, p_3 – три точки множини S, що визначають коло $C(v)$. Якщо коло містить ще деяку точку $p_4 \in S$, то вершина v знаходиться ближче до p_4, ніж до інших точок. Тоді v має знаходитися у $V(4)$, що суперечить тому, що v належить одночасно $V(1)$, $V(2)$ та $V(3)$.

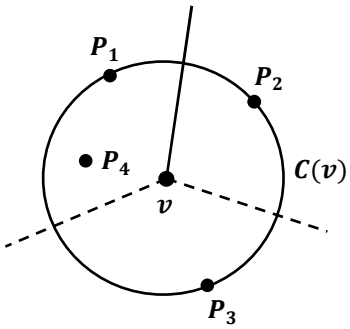


Рисунок 185. Для будь-якої вершини діаграми Вороного коло не містить інших вершин

Теорема 40. Кожний найближчий сусід точки $p_i \in S$ визначає ребро в многокутнику Вороного $V(i)$ (рис. 186).

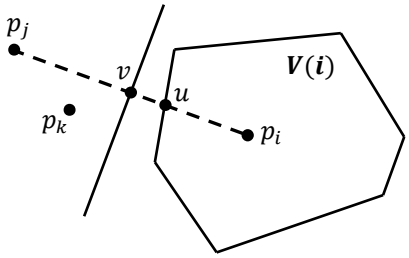


Рисунок 186. Найближчий сусід точки на діаграмі визначає ребро в многокутнику Вороного

Нехай p_j – найближчий сусід p_i, а v – середина відрізку, що їх з'єднує. Нехай v не лежить на границі $V(i)$. Тоді відрізок p_iv перетинає деяке ребро многокутника $V(i)$ (наприклад, рівновіддалене від p_i та p_k) в деякій точці u.

Тоді $|p_iu| < |p_iv|$ і тому $|p_ip_k| \leq 2|p_iu| < 2|p_iv| = |p_ip_j|$, звідки випливає, що p_k ближче до p_i ніж p_j, що суперечить умові теореми.

Теорема 41. Многокутник $V(i)$ є необмеженим тоді і тільки тоді, коли точка p_i лежить на границі опуклої оболонки множини S.

Теорема 42. Граф, двоїстий діаграмі Вороного, є тріангуляцією множини S (тріангуляція Делоне) (рис. 187).

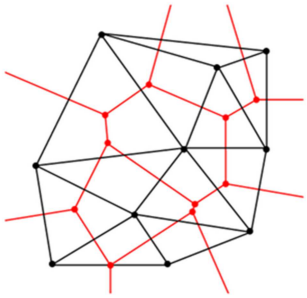


Рисунок 187. Тріангуляція Делоне

Теорема 43. Діаграма Вороного множини з N точок має не більше $2N-5$ вершин та $3N-6$ ребер.

4.3.3 Методи побудови діаграми Вороного

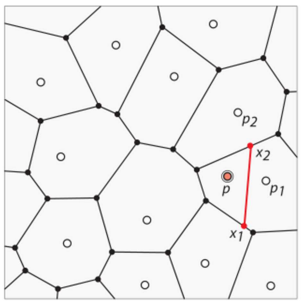
Для побудови діаграми можна безпосередньо *використати означення*: кожен многокутник Вороного шукається окремо як перетин $N-1$ півплощин. Найкращий алгоритм для цього випадку дасть час $O(N^2 \log N)$.

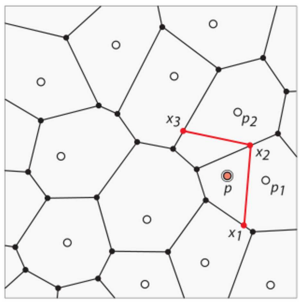
Простим і через це досить популярним є **інкрементний алгоритм** (Грін, Сібсон). Його складність $O(N^2)$). Однак рандомізована версія працюватиме за очікуваний час $O(N \log N)$.

Ідея полягає в тому, щоб маючи діаграму Вороного $Vor(S)$ для точок $p_1, \ldots, p_k$ певним чином модифікувати її при додаванні нової точки p.

Діаграма Вороного є розбиттям площини, тому нова точка p потрапить до деякого многокутника $V(i)$ (самостійно подумайте над ситуацією, коли p лежить на ребрі чи у вершині діаграми).

Серединний перпендикуляр до p_1p перетинає границю $V(1)$ рівно в двох точках x_1 та x_2. Нехай трійка точок px_1x_2 впорядкована проти стрілки годинника. Відрізок x_1x_2 розбиває комірку $V(1)$ на дві частини, одна з яких буде належати шуканому $V(p)$ (рис. 188а).



(а)


(б)

Рисунок 188. Ілюстрація інкрементного алгоритму

За допомогою x_1x_2 отримаємо решту границі $V(p)$. Точка x_2 лежить на границі многокутників $V(1)$ та $V(2)$. Розглянемо тепер серединний перпендикуляр до p_2p і отримаємо відрізок x_2x_3, який розбиває комірку $V(2)$ на дві частини, одна з яких буде належати $V(p)$.

Процес продовжиться, поки не досягнемо x_1, таким чином зібравши по сегментам всю границю $V(p)$. В кінці потрібно вилучити ті частини діаграми Вороного, що потрапили всередину $V(p)$ (рис. 188б).

Розглянемо покрокову ілюстрація інкрементного алгоритму побудови діаграми Вороного (рис. 189).

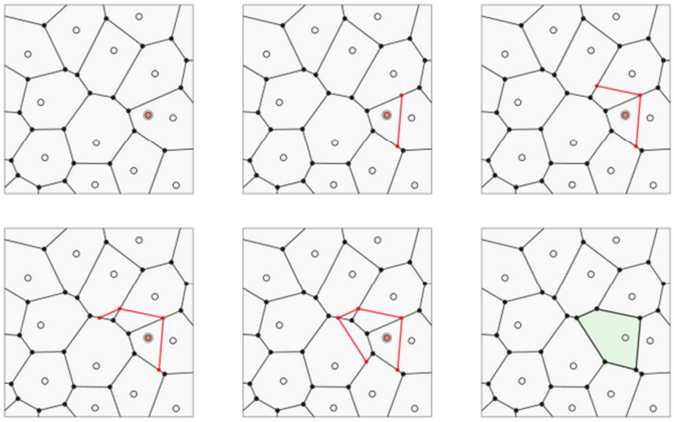


Рисунок 189. Покрокова ілюстрація інкрементного алгоритму побудови діаграми Вороного

Діаграма Вороного є планарним графом. Вона може бути представлена реберним списком з подвійними зв’язками.

Теорема 44. Для побудови діаграми Вороного множини з N точок необхідно $\Omega(N \log N)$ операцій в найгіршому випадку.

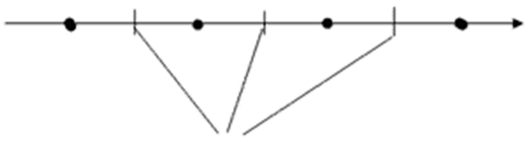


Рисунок 190. Вершини діаграми Вороного – одновимірний випадок

Наслідок 2. СОРТУВАННЯ ∞_N ДІАГРАМА ВОРОНОГО.

Схема «розділяй та владарюй»

Крок 1. Розділити множину S на дві приблизно рівні підмножини S_1 та S_2.

Крок 2. Рекурсивно побудувати $Vor(S_1)$ та $Vor(S_2)$.

Крок 3. Об'єднати $Vor(S_1)$ та $Vor(S_2)$ і таким чином отримати $Vor(S)$.

Нехай для заданого розбиття $\{S_1, S_2\}$ множини S $\sigma\{S_1, S_2\}$ означає множину ребер діаграми Вороного, спільних для пар многокутників $V(i)$ та $V(j)$ діаграми $Vor(S)$, де $p_i \in S_1$ та $p_j \in S_2$.

Теорема 45. Сукупність $\sigma(S_1, S_2)$ є множиною ребер деякого підграфа діаграми $Vor(S)$ і має такі властивості:

- $\sigma(S_1, S_2)$ складається із циклів та ланцюгів, що не мають спільних ребер. Якщо деякий ланцюг містить єдине ребро, то це ребро є прямою лінією; інакше два крайні ребра ланцюга є променями;
- якщо множини S_1 та S_2 є лінійно роздільними, то $\sigma(S_1, S_2)$ складається з єдиного монотонного ланцюга (при цьому якщо розділяючій прямій належить декілька точок, вони мають належати до однієї множини розбиття). Якщо розділяюча пряма вертикальна, то σ розділяє площину на ліву π_L та праву π_R частини.

Теорема 46. Якщо множини S_1 та S_2 лінійно розділені вертикальною прямою і при цьому S_1 міститься лівіше від S_2, то діаграма Вороного $Vor(S)$ є об'єднанням $Vor(S_1) \cap \pi_L$, $Vor(S_2) \cap \pi_R$ та σ.

procedure ДІАГРАМА ВОРОНОГО

Крок 1. Розділити множину S на дві приблизно рівні підмножини S_1 та S_2 за медіаною по x-координаті.

Крок 2. Рекурсивно побудувати $Vor(S_1)$ та $Vor(S_2)$.

Крок 3'. Побудувати ламану σ, що розділяє S_1 та S_2.

Крок 3". Вилучити усі ребра діаграми $Vor(S_2)$, що зліва від ланцюга σ, та всі ребра $Vor(S_1)$, що справа від σ, отримавши $Vor(S)$.

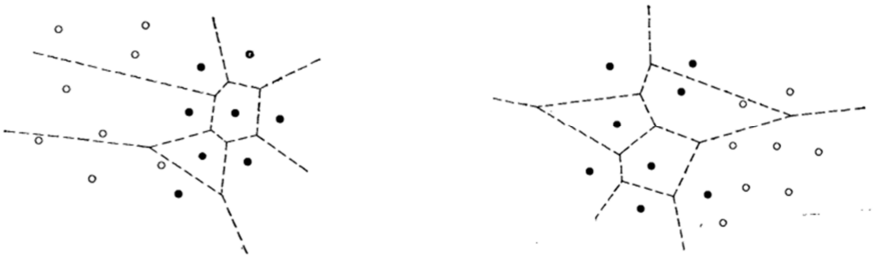


Рисунок 191. Діаграми Вороного правої та лівої підмножини

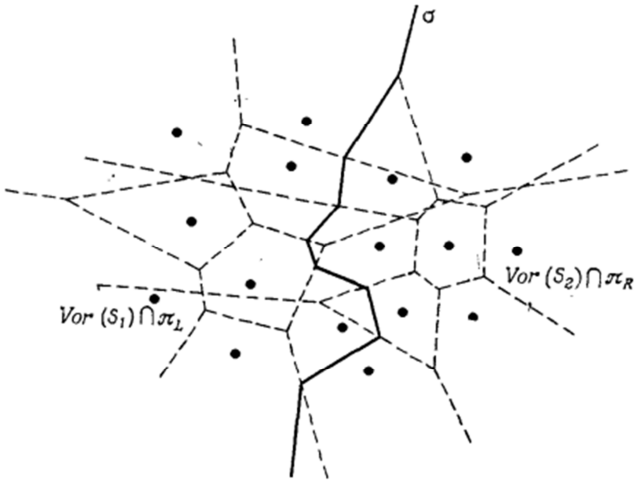


Рисунок 192. Накладання $Vor(S_1)$, $Vor(S_2)$ та σ

Пошук медіани можна здійснити за час $O(N)$. На вилучення ребер на кроці 3" піде час $O(|S_1| + |S_2|) = O(N)$. Для загальної складності алгоритму $\Theta(N \log N)$ слід провести побудову розділяючого ланцюга за $O(N)$.

Побудова розділяючого ланцюга

Кожний промінь ланцюга σ перпендикулярний опорному відрізку до $CH(S_1)$ та $CH(S_2)$ і ділить його навпіл. S_1 та S_2 за припущенням лінійно роздільні, тому існує рівно два опорних відрізки до $CH(S_1)$ та $CH(S_2)$.

Знайшовши промінь ланцюга σ, послідовно будуємо ребра ланцюга до досягнення другого променя (рис. 193).

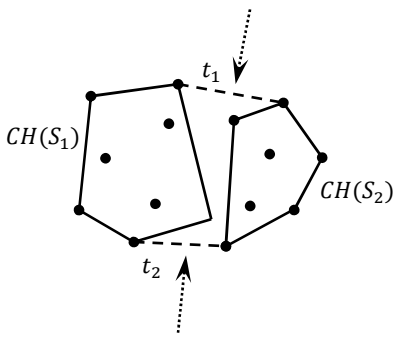


Рисунок 193. Знаходження променя ланцюга σ та побудова відповідного ребра

Будуємо множину ребер розділяючого ланцюга між вхідним та вихідним променями. Верхній промінь σ перпендикулярний до опорного відрізка 7–14 і ділить його навпіл. Уявімо точку z, яка, рухаючись по верхньому променю, перетне ребро діаграми $Vor(S_2)$. Отже точка z переходить із області близькості точки 14 в область близькості точки 11, тобто стає ближчою до 11 ніж до 14. Тому промінь e_2 буде перпендикулярним відрізку 7–11.

Рухаючись по променю e_2, точка z перетне ребро $Vor(S_1)$, тобто вона перейде із області близькості точки 7 у область близькості точки 6, тому e_3 стає перпендикулярним відрізку 6–11, а тоді e_4 – відрізку 6–10, і так далі.

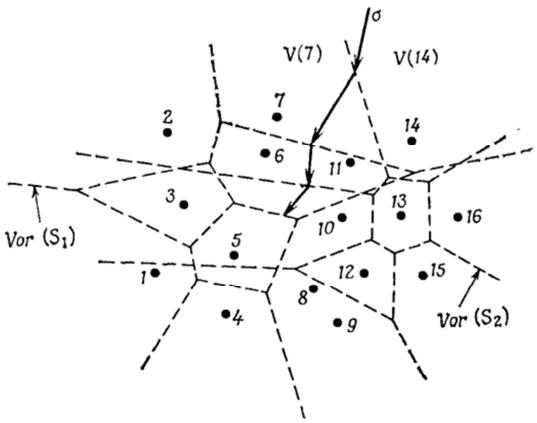


Рисунок 194. Побудова розділяючого ланцюга

Для визначення точок перетину ланцюга σ з $V(i)$ у $Vor(S_1)$ необхідно переглядати ребра за годинниковою стрілкою (рис. 195). Для σ з $V(j)$ у $Vor(S_2)$ – навпаки.

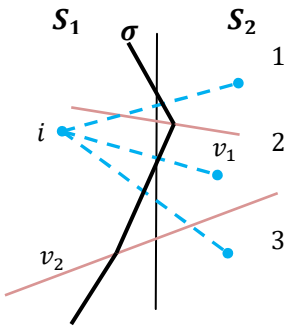


Рисунок 195. Визначення точок перетину ланцюга

Позначимо:

- $I(e, e')$ – перетин відрізків e і e';
- $I(e, e') = \wedge$ означає, що e і e' не перетинаються;
- t_1 і t_2 – два опорних відрізки, при цьому $t_1 = [p, q]$.

Розглянемо реалізацію кроку 3' процедури *діаграма Воронова*.

begin
$p_L := p;\ p_R := q;\ e := e^*;\ v := v^*;$
$e_L :=$ перше ребро (відкритої) границі $V(p_L)$;
$e_R :=$ перше ребро (відкритої) границі $V(p_R)$;
repeat
while $(I(e, e_L) = \Lambda)$ **do**
$e_L := \mathrm{H}_1[\mathrm{e}_L]$; (*подивитись границю $V(p_L)$*)
while $(I(e, e_R) = \Lambda)$ **do**
$e_R := \mathrm{H}_2[\mathrm{e}_R]$; (*подивитись границю $V(p_R)$*)
if (v ближче до $I(e, e_L)$, ніж до $I(e, e_R)$) **then**
begin
$v: = I(e, e_L)$;
$p_L :=$ точка S, яка міститься по іншу сторону від e_L (сусід (p_L));
$e: =$ пряма, перпендикулярна $[p_L, p_R]$, і ділить його навпіл;
$e_L :=$ обернене до e_L (*нове e_L є ребром $V(p_L)$*);
end
else
begin
$v: = I(e, e_R)$;
$p_R :=$ точка S, яка міститься по іншу сторону від e_R (сусід (p_R));
$e: =$ пряма, перпендикулярна $[p_L, p_R]$, і ділить його навпіл;
$e_R :=$ обернене до e_R
end
until $([p_L, p_R] = t_2)$
end.

Алгоритм Форчуна (Fortune)

Метод працює за час $\mathrm{O}(N \log N)$ і є простішим за підхід на основі «розділяй та владарюй». В основі алгоритму – метод плоского замітання.

Нехай пряма замітає площину згори вниз. В пройденій частині маємо діаграму Вороного для тих точок, що лежать вище замітаючої прямої. Головна проблема полягає в тому, що точки, які знаходяться *перед* замітаючою прямою, можуть породжувати вершини діаграми, що лежать *за* цією прямою (рис. 196).

Позиції відмічених вершин діаграми Вороного залежать від точок, які ще не пройшла замітаюча пряма і, отже, невідомі алгоритму.

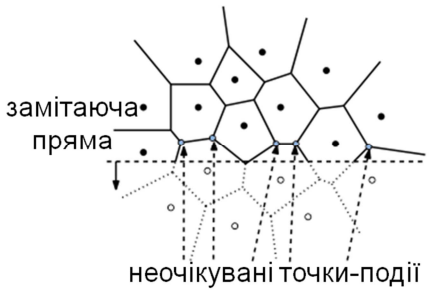


Рисунок 196. Метод плоского замітання в алгоритмі Форчуна

Вихід: будується додаткова «межа», що відокремлює область, в якій точно вже не станеться змін (тобто не з'явиться вершина діаграми Вороного).

В заметеній частині можна виділити «берегову лінію» – множину точок, рівновіддалених від замітаючої прямої і найближчої від них точки $p \in S$.

Геометричне місце точок, рівновіддалених від заданої точки, що лежить над прямою, і цієї прямої – парабола. Таким чином, берегова лінія є послідовністю параболічних дуг. Вона буде монотонна по осі абсцис.

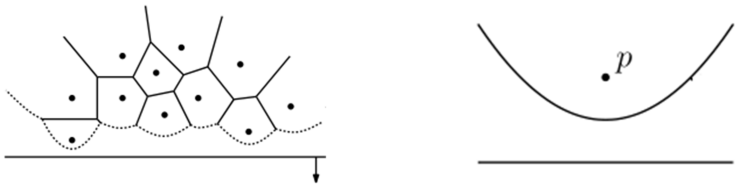


Рисунок 197. Геометричне місце точок, рівновіддалених від заданої точки є параболою

Будемо відслідковувати зміни в береговій лінії: моменти появи і зникнення дуг на ній. Точки перетину дуг («брейкпоінти») парабол лежать на ребрах діаграми Вороного. Берегова лінія зберігається у вигляді збалансованого двійкового дерева пошуку.

Зауваження. При цьому явним чином ми не зберігаємо ні самі параболи, ні точки їх перетину.

Події будуть двох типів:

- події точки (замітаюча пряма зустріла нову точку; до берегової лінії додається дуга);
- події кола (з берегової лінії видаляється дуга при збігу брейкпоінтів; в цій точці з'являється нова вершина діаграми Вороного).

Подія точки. В момент додавання нової точки відповідна дуга берегової лінії вироджена в промінь. При русі замітаючої прямої промінь розширюється в параболу на береговій лінії. При цьому з'являються два нових брейкпоінти, які в початковий момент часу співпадають, а потім починають рухатися в протилежних напрямках (рис. 198).

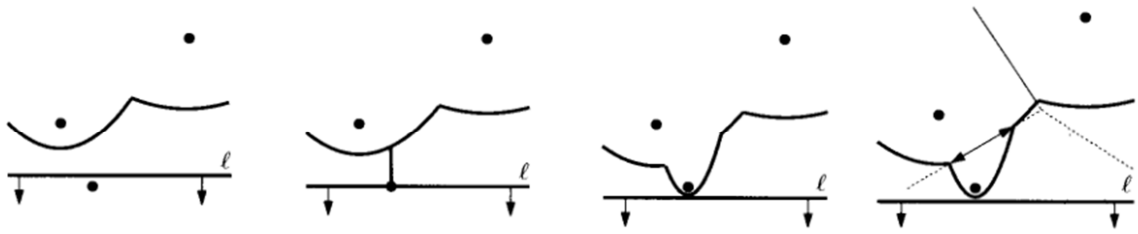


Рисунок 198. Додавання нової точки

Це єдиний спосіб додавання дуги до берегової лінії. Берегова лінія складається не більше ніж з $2N-1$ параболічних дуг.

Подія кола. Ця подія створюється динамічно. Вона генерується трьома сусідніми точками на береговій лінії. Інших способів зникнення дуги з берегової лінії немає.

В момент, коли дуга α' зникає, всі 3 параболи проходять через одну точку q; вона рівновіддалена від точок p_i, p_j, p_k та прямої l (рис. 199).

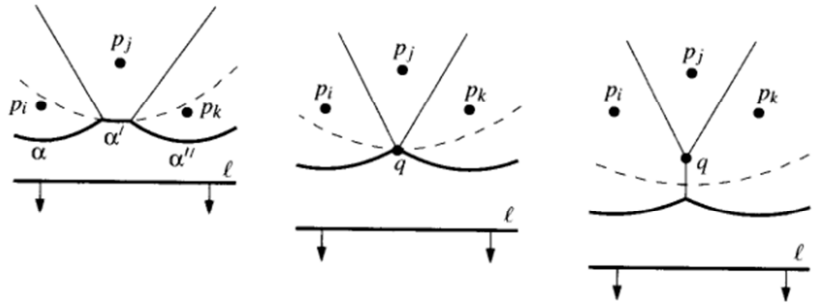


Рисунок 199. Зникнення дуги з берегової лінії

4.3.4 Діаграма Вороного та задачі про близькість

Теорема 47. Діаграму Вороного на множині із N точок площини можна побудувати з оптимальним часом $\Theta(N \log N)$.

Теорема 48. Задача *всі найближчі сусіди* зводиться за лінійний час до задачі *діаграма Воронова*, і тому її можна розв'язати за оптимальний час $\Theta(N \log N)$.

Теорема 49. Задача *найближча пара* зводиться за лінійний час до задачі *діаграма Воронова*, і тому її можна розв'язати за оптимальний час $\Theta(N \log N)$.

Теорема 50. Пошук найближчого сусіда можна виконати за оптимальний час $O(\log N)$, використовуючи пам'ять об'ємом $O(N)$, з витратами на попередню обробку (побудова діаграми Вороного) $O(N \log N)$.

Теорема 51. Якщо для множини точок побудована діаграма Вороного, то опуклу оболонку цієї множини можна побудувати за лінійний час.

Теорема 52. Триангуляцію, в якій коло, описане навколо будь-якого трикутника, не містить інших точок, можна побудувати за оптимальний час $\Theta(N \log N)$. Це триангуляція Делоне.

Рисунок 200. Зв'язок задач про близькість з основними задачами,
які використовуються як обчислювальні прототипи

4.3.5 Триангуляція Делоне

Триангуляція Делоне в загальному випадку відповідає прямолінійному двоїстому графу до діаграми Вороного (рис. 201). Особливими випадками є: три точки на прямій, чотири точки на колі.

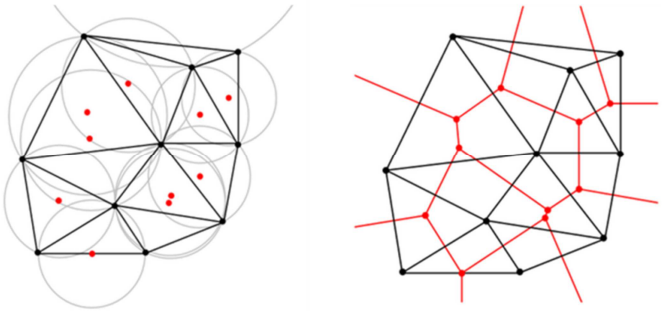


Рисунок 201. Триангуляція Делоне та відповідний прямолінійний двоїстий граф

Коло, описане навколо будь-якого трикутника, не містить інших точок.

Триангуляція Делоне максимізує найменший кут. Найменший кут буде не меншим ніж в будь-якій іншій триангуляції. При цьому триангуляція Делоне не обов'язково мінімізує максимальний кут чи довжину ребер (рис. 202).

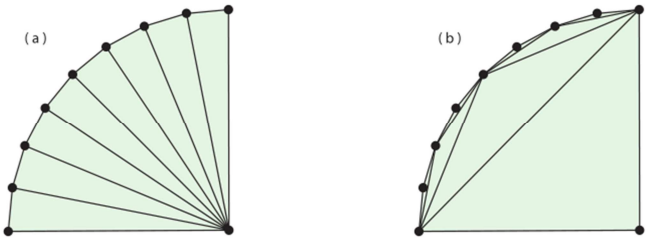


Рисунок 202. (a) Триангуляція Делоне, (b) інша триангуляція

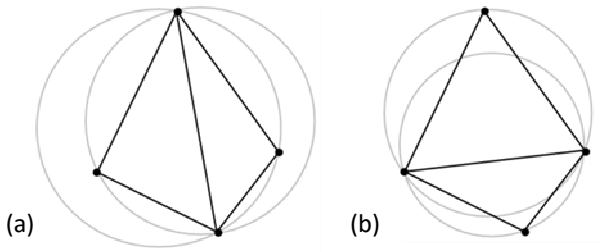


Рисунок 203. (a) Не є триангуляцією Делоне, (b) Триангуляція Делоне

Можна помітити, що мінімальний кут збільшився.

Прямі методи побудови триангуляції Делоне:

- метод заміни ребра;
- інкрементний алгоритм;
- підхід «розділяй та владарюй».

ПЕРЕЛІК ПИТАНЬ ДЛЯ САМОКОНТРОЛЮ

1. Основні класи та типи задач обчислювальної геометрії. Сфери застосування.
2. Поняття оцінки складності алгоритму.
3. Метод звідності задач. Визначення верхніх та нижніх оцінок (твердження).
4. Задачі-прототипи. Нижні оцінки складності задач.
5. Загальні означення обчислювальної геометрії.
6. Структури даних. Множини, списки, черги.
7. Дерево відрізків. Операції вставки та вилучення інтервалів.
8. Реберний список з подвійними зв'язками.
9. Метод плоского замітання.
10. Поняття геометричного пошуку. Міри ефективності. Моделі геометричного пошуку. Типи пошукових запитів.
11. Метод векторного домінування. Метод локусів.
12. Локалізація точки на планарному розбитті. Розв’язання задачі про приналежність простому многокутнику та опуклому многокутнику.
13. Метод смуг. Метод деталізації триангуляції. Метод трапецій. Оцінки складності. Переваги та недоліки.
14. Метод ланцюгів. Оцінки складності. Регуляризація графа.
15. Регіональний пошук. Основні типи дій. Оцінки складності. Двовимірний випадок.
16. Регіональний пошук. Метод 2-d дерева. Метод дерева регіонів та використання техніки fractional cascading для покращення методу.
17. Регіональний пошук у просторах вищих розмірностей.
18. Опуклі оболонки. Основні поняття. Постановка та схема розв’язання основних задач. Нижня оцінка задачі побудови опуклої оболонки множини точок.
19. Метод Грехема та метод Джарвіса. Оцінка складності. Переваги та недоліки. Метод Чана: особливості.
20. Швидкий метод побудови опуклої оболонки («ШвидкОбол»). Оцінка складності. Переваги та недоліки
21. Методи побудови опуклої оболонки типу "розділяй та владарюй". Оцінка складності.
22. Динамічні алгоритми побудови опуклої оболонки. Відкритий алгоритм Препарати. Інкрементний алгоритм. Оцінки складності.
23. Алгоритм динамічної підтримки опуклої оболонки. Оцінка складності.
24. Алгоритм апроксимації опуклої оболонки. Оцінка складності.
25. Опукла оболонка простого многокутника (алгоритм Лі). Оцінка складності.
26. Узагальнення алгоритмів побудови опуклої оболонки на вищі розмірності.
27. Метод «загортання подарунку», «розділяй та владарюй» та рандомізований інкрементний алгоритм в 3D.
28. Близькість. Постановка основних задач. Обгрунтування їх нижніх оцінок складності.
29. Найближча пара - метод «розділяй та владарюй».
30. Означення та властивості діаграми Вороного.
31. Методи побудови діаграми Вороного: інкрементний алгоритм, метод Форчуна, метод «розділяй та владарюй» та їх складність.
32. Розв’язання задач близькості за допомогою діаграми Вороного.
33. Триангуляція Делоне, її властивості. Зв'язок з діаграмою Вороного.
34. Перетин. Постановка основних задач.
35. Пошук перетинів відрізків методом плоского замітання.
36. Перевірка перетину простих многокутників та простоти многокутника.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Ф. Препарата, М. Шеймос. Вычислительная геометрия: введение. – М.: Мир, 1989.
2. А. Ахо, Дж. Хопкрофт, Дж. Ульман. Построение и анализ вычислительных алгоритмов. – М.:Мир, 1979.
3. В.М. Терещенко, І.В. Кравченко, А.В. Анісімов. Основні алгоритми обчислювальної геометрії. – К., 2002.
4. S.L Devadoss, J. O'Rourke. Discrete and Computational Geometry. Princeton University Press, 2011.
5. M. de Berg, O. Cheong, M. van Kreveld, M. Overmars. Computational Geometry: Algorithms and Applications. 3rd edition. – Springer, 2008.
6. J. O'Rourke. Computational Geometry in C. Cambridge University Press, Second Edition, 1998.
7. David M. Mount. Lecture notes for the course CMSC 754 Computational Geometry (https://www.cs.umd.edu/class/fall2014/cmsc754/Lects/cmsc754-fall14-lects.pdf)
8. Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Стайн. Алгоритмы – построение и анализ. Третье издание. – М.: ИД "Диалектика-Вильямс", 2019.
9. М. Ласло. Вычислительная геометрия и компьютерная графика на C++. – М.: Бином, 1997.